CLIPSBASE: A REAL-TIME RELATIONAL DATABASE FOR THE "PRINCIPAL INVESTIGATOR-IN-A-BOX" [PI] EXPERT SYSTEM

by

Sen-Hao Lai

S.B., Aeronautics and Astronautics
Massachusetts Institute of Technology
(1989)

Submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE
in
AERONAUTICS AND ASTRONAUTICS
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1991

Signature of Author [illegible]

Certified by [illegible]
Laurence R. Young
Professor of Aeronautics and Astronautics

Certified by [illegible]
Peter Szolovits
Associate Professor of Electrical Engineering and Computer Science

Accepted by [illegible]
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

CLIPSBASE: A REAL-TIME RELATIONAL DATABASE FOR THE "PRINCIPAL INVESTIGATOR-IN-A-BOX" [PI] EXPERT SYSTEM

by

Sen-Hao Lai

Submitted to the Department of Aeronautics and Astronautics in partial fulfillment of the requirements for the Degree of Master of Science in Aeronautics and Astronautics

ABSTRACT

In many real-time expert systems, a custom-built Database Management System (DBMS) is needed to manipulate data in an organized and effective manner. This research concludes that a unique approach, by implementing a database in an expert system shell, can indeed make this data manipulation more effective. Several design considerations are presented that apply to a developer who wishes to build DBMS functionality into a rule-based representation scheme. These considerations include host language constraints, data models, real-time constraints, and common real-time mechanisms.

An analysis of various access mechanisms is given, specifying how they can be used more effectively. Also discussed is how computational overhead is to some extent dependent on how the data model is implemented in the knowledge base. The real-time mechanisms described include query optimization, shared processes and results, and minimization of file access. Their effectiveness is determined by comparison to access schemes in a distributed file system. The goal of this thesis is to provide DBMS capability to a knowledge shell so that data and knowledge can interact more efficiently.

Thesis Supervisor: Dr. Laurence R. Young

Title: Professor of Aeronautics and Astronautics

Thesis Co-Supervisor: Dr. Peter Szolovits

Title: Associate Professor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

First, I would like to thank Peter Szolovits and Larry Young for their suggestions and guidance in the past year and a half. Their wisdom and advice are indispensable for those doing research in expert systems during their pursuit of an advanced degree at M.I.T. As with any graduate level study, it is always helpful to have a mentor or two provide a guiding light toward that proverbial end of the tunnel.

Second, special thanks go to Rajiv Bhatnagar for putting up with all my stupidity and opening up my eyes to the field of Artificial Intelligence. It was Rajiv who encouraged me to attend the AAAI conference during the summer of 1990. I felt like a wide-eyed kid in a first grade field trip, saying "wow" to everything and taking keen interest in all the marvels of modern science and engineering. It was also Rajiv who suggested that I do a thesis in expert system databases. Through his continuous encouragement, feedback, and recommendations, CLIPSBase was born.

Third, thanks go to my friends at the Man-Vehicle Lab, especially Nick Groleau, with whom I shared an office, much conversation, and endless humor. I will not forget the good times I had with Jock, Glenn, Keoki, Tom, Brad, and a host of other wonderful characters with whom I spent a crazy weekend rafting, camping, and hiking in Maine.

Fourth, I must extend thanks to the [PI] team, including Rich Frainier, Silvano Colombano, Meera Manahan, Mike Compton, Chih-Chao Lam, Jurine Adolf, Tina Holden, and Irv Statler, with whom I spent a countless number of hours in telecon and exchanging email.

Fifth, I thank NASA for writing CLIPS whose piece of software, for better or worse, controls the center of the [PI] knowledge base, and for not cutting funding while I was pursuing my Master's at M.I.T.

Sixth and final, deepest gratitude go to Huiying Li, whose support during the past year has made my graduate studies a more bearable experience.

This research was co-funded by the NASA Space Station Freedom Advanced Development Program and the Office of Aeronautics, Exploration, and Technology AI Program.

CONTENTS

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 Background

All expert systems consist of two primary components: an inference engine with which reasoning may be performed and a knowledge base containing the facts and heuristics on which reasoning may be applied. The inference engine uses a certain knowledge representation, usually in the form of rules (also known as production rules) or units (also called frames, schemas, or list structures). [9]

Rules come in only one simple syntactical structure: the *IF/THEN* statement. [17] For example, Rule 1: *IF Joe is late to work THEN he is out of a job.* The rule consists of a conditional predicate, which if true, establishes a conclusion from causal relationship. By chaining a series of *IF/THEN* statements, certain aspects of a complicated process of the human mind known as reasoning can be duplicated in a computer program:

Rule 2: *IF Joe forgets to set his alarm clock THEN he will oversleep.*

Rule 3: *IF Joe oversleeps THEN he will miss his ride.*

Rule 4: *IF Joe misses his ride THEN he will be late to work.*

By putting the four rules together, a computer program follows the logic to conclude that if Joe forgets to set his alarm clock, he will be out of a job.

Units are a more passive way of representing knowledge. Such a representation scheme is based on the relationships between entities and properties associated with each entity. Typically, a unit consists of a list of values associated with those properties. For example, "a car has an engine system, electrical system, and brake system." Because in a task domain, the entities must stand in various relations with each other, links can be established between the units. The carburetor is part of the fuel system, which is part of the engine system, and so forth. [17]

The knowledge base contains the facts and heuristics on which the rules may be applied. This includes factual knowledge commonly accepted by experts in the task domain, which may come from journals and textbooks. It also includes heuristic knowledge, or "rules of thumb." This type of knowledge is more a priori and experimental, being based on the good judgement of an expert in the field. Heuristics encompass the problem-solving know-how of someone who is able to deal efficiently with problems in the task domain and who can provide good guesses. It is the heuristics which give the expert a problem-solving edge. [17]

A subset of the knowledge base is the database. The database contains pieces of information from which the facts and heuristics can be built. It also organizes this information in a form that can be recognized by the knowledge base so that data manipulation can be performed. Data manipulation can mean, among many things, information retrieval, storage, modification, and removal. Within the database, there is a Database Management System (DBMS) consisting of various functions that allow the knowledge in an expert system to interact with the data more effectively.

What gives an expert system its power is the amount of knowledge it contains. The more knowledge it has, the smarter it is at solving problems in the task domain. Therefore, the most important ingredient in an expert system is the high-quality knowledge residing in its inference engine and knowledge base. [17] To this end, knowledge engineers have built expert systems consisting of thousands of rules and enormous databases to house all the information needed as part of the knowledge base. Most important, as the database grows, the knowledge base grows. As the knowledge base grows, the system gets smarter. It is for this very reason that expert systems also consist of a database to encompass much of the data needed in making decisions. The database may contain historical data which the system accumulates to allow statistical analysis, or it may contain temporary variables as intermediary information for other calculations.

To illustrate, the Nippon Life, Japan's largest life insurance company, receives some 2.7 million life insurance applications a year. Of these, 800,000 applications must be evaluated by their expert system called Underwriter's Aid, developed by Computer

Services Kaisha (CSK) using Intellicorp's KEE expert system shell, based on medical data and mortality statistics related to a compendium of diseases. In order to evaluate each application, the expert system must assess the data provided by each applicant and analyze it against the information related to previous life insurance policies. All this medical and statistical data reside in IBM data-processing mainframes, one of the largest in installations in Japan. The advisory applications for the expert system run on microcomputers linked directly to the mainframe complex. [9] Without a database, the evaluations simply could not be made.

Likewise, when Britain's social security system needed an expert system to offer a Retirement Pension Forecasting Service to the citizens, Arthur Anderson built a prototype called Pension Advisor consisting of 330 rules to navigate the maze of Britain's social security rules and regulations and help pensioners handle a variety of inquiries and circumstances related to a citizen's Basic Pension and Additional Pension plans, including change in retirement schedule, payment of debts, insurance existing overseas, and much more. When a citizen's question arrives, the citizen's record is retrieved from a mainframe containing the database. The pensioner then sets to work with the expert system residing on a microcomputer. It is therefore a two-step process. Within minutes, an answer is provided for the citizen's query. [9] Although the database is not physically part of the expert system's knowledge base, it is clear that the information within the database is critical to the system's decision-making process.

1.2 The Problem

Because the database is such an important part of the knowledge base, expert system shell vendors incorporate some facility for manipulating and storing data in their systems, and/or they provide access mechanisms to existing database applications to eliminate the need for writing hooks. For example, Gold Hill Computers' GoldWorks II allows direct access to dBase, a data processing application, without requiring the knowledge engineer to write special code. GURU, from MDBS Inc., has a built-in complete SQL language interface, an internal relational database management system, and spreadsheet. Furthermore, it provides hooks to external sequential files, mainframe Database Management Systems (DBMS'), PC DBMS', and PC spreadsheets. [11]

For real-time applications, however, many of these commercial tools are inappropriate for a number of reasons. The system runs too slowly, or the architecture does not allow the inference engine to directly access the database, or the access mechanisms are too inefficient for real-time purposes, or the DBMS facilities come with a great deal of unnecessary overhead, or funding is insufficient to purchase such a sophisticated yet expensive piece of software. For one or more of these reasons, knowledge engineers usually rely on a custom-made DBMS, a feature that is quite common in the artificial intelligence community. [10] Moreover, if the last reason is applicable, they get a more economical development environment like CLIPS, a rule-based expert system shell that provides neither a database nor hooks to external database applications, thereby forcing the knowledge engineers to develop their own custom DBMS.

Although real-time applications have been built using an expert system shell with built-in storage functions and hooks to external databases, it is in many cases simply more appropriate to write a custom database. The problem may be that the architecture of the expert system imposes too many storage and retrieval problems for the application's real-time purposes, or access to an external database cannot be achieved directly or efficiently, or the shell simply does not provide a good enough DBMS, or undesirable overhead creeps into the system by using external DBMS facilities. For the American Express Authorizer's Assistant expert system, some of these problems became true. For the "Principal Investigator-in-a-Box" [PI] expert system currently under development, all of these problems are true.

1.2.1 The American Express Approach

When a buyer goes to make a purchase at a store using an American Express Card, the sales clerk takes the amount of purchase, reads the magnetic strip, and sends the information over telephone lines to one of two operating centers at Phoenix, Arizona or Brighton, England. At one of these operating centers, some statistical analysis takes place and returns the approval to the point-of-sale terminal in front of the sales clerk. If a questionable case arises, the transaction is changed over to human authorizers at one of the authorizing centers such as the Fort Lauderdale center. Here, the human authorizer is presented with the transaction under question and has seventy seconds to decide to approve or reject the purchase. The conventional way of making such a decision was to analyze the individual's credit record, maybe talk to the merchant or purchaser, and then decide yes or no by applying a large body of knowledge, policies, and rules accumulated over years of experience American Express has had with these transactions. This procedure may include consulting a four to five inch authorizers' training manual. [9]

The Authorizers' Assistant (AA) expert system was built to help authorizers with this high pressure, complicated, and demanding task of deciding whether or not to authorize a transaction, and at the same time reduce the losses due to poor judgements. The AA system was developed by Inference Corp. using ART to run on a Symbolics Lisp machine, piggybacking twelve databases on an IBM mainframe. From the beginning, this architecture imposed many problems in storage and retrieval. Worse, the expert system does not access the information directly from the databases. The system gets the data by first retrieving the screens that an authorizer would normally see in a transaction and then parsing the information before being digested by the knowledge base. Problems crept in during the development of AA. Errors and extra overhead appeared in the database systems. Even the project manager, Bob Flast, admitted that this was a suboptimal solution. [9]

1.2.2 The "Principal Investigator-in-a-Box" [PI] Approach

The expert system with which the author is personally involved is called "Principal Investigator-in-a-Box" [PI], or more formally, Astronaut Scientific Advisor. The purpose of [PI] is to advise astronauts on the Space Shuttle during scientific experiments. This advice usually takes the form of information required in trouble-shooting a malfunctioning piece of equipment or changes in the experiment's schedule. [2,20]

The motivation for this system was that, despite their rigorous training and scientific backgrounds, astronauts are often unprepared to handle all the contingencies and unexpected events that may occur during an in-flight experiment. Although the astronaut can sometimes rely on communication with the experiment's Principal Investigator (PI), who is on the ground when unexpected circumstances occur, this spacecraft-to-ground communication is often not timely enough or is of insufficient bandwidth to permit the PI to effectively assist the astronauts. Hence, the [PI] expert system is being developed such that it would be capable of performing rapid data analysis and providing recommendations to the astronauts that the PI would provide if he were available on board during the experiment. [2,20]

The [PI] system is being developed using CLIPS, from the Artificial Intelligence Section at NASA/JSC, and HyperCard on the Macintosh platform. It currently consists of eight different software modules on two Macintosh IIx's to collect and reduce raw data from an experiment, monitor an experiment's progress and suggest modifications to the schedule when necessary, recognize experimental data that are likely to be "interesting", diagnose and correct problems in the experimental equipment when they occur, generate new experiments based on previous "interesting" results, moderate inter-module communication and properly allocate system resources, and allow astronauts to interact through an interface. [2,20] Of these modules, three are knowledge bases already in operation. At present the setup uses an ad hoc distributed file system to store its data. There are four files, each with its own file structure, to store the processing results of the three knowledge base modules residing in the CLIPS shell, and to store the information contained in an inter-module communication software residing in the HyperCard application. Such a file structure was used because the CLIPS shell does not provide a built-in DBMS. The problems of this architecture are readily apparent.

When one of the knowledge base modules wishes to retrieve historical data needed for its processing, it must first send a message to the communication module requesting the piece of information. The communication module interprets the message and retrieves the information from its own file or one of the other files and sends the data to the inquiring knowledge base module. In this set-up, much time is wasted by first asking for the data, having its request interpreted, and then getting the information from outside its shell environment. Furthermore, updating data for concurrency is difficult. Four different update mechanisms for four different file formats are used. Although historical data are stored in the HyperCard environment, this information is not directly accessible by those knowledge bases residing in CLIPS. As a result, some of the information needed in the reasoning is requested from the user even though the information already exists in one of the files. This indirect information retrieval scheme is obviously not the optimal solution.

An alternative was to use a commercial database application. This was one of the considerations during the project's initial development stages. Unfortunately, commercial databases usually come with a large DBMS. The DBMS provides facilities for "concurrent" access, update, protection, and security of the database. For systems, like [PI], which have only a small concurrency requirement and do not need much data security, an extensive DBMS is of little use and causes unnecessary overhead. Also, a commercial database not compatible with the expert system language and paradigm (e.g. rule-based, object-oriented, model-based, etc.) involves significant overhead needed in communicating with the modules of that expert system. Worse, most commercial databases require that they run as single, independent applications, allowing no other application to run while a database is being used. Because of these factors, the project team realized that [PI] does not need all the facilities of a commercial database, as the needs of [PI] are small scale, and the overhead of a general purpose database would consume extensive computing resources that would compromise the performance of the expert system.

1.3 A Better Solution

The solution that the author finally chose to implement for [PI] is based primarily on the real-time constraints of the expert system and the fact that some DBMS capabilities are needed in order to store and retrieve the data in an organized fashion. The database must provide some of the functions of a commercial database, but not all. At the same time, it must perform access operations quickly enough to fall within the real-time constraints. More importantly, the data must be directly accessible to the knowledge base modules in order avoid the unnecessary overhead and waste in time associated with the distributed file system. This means that centralization of the data is required.

The scheme chosen is to implement the database entirely within the CLIPS shell. A relational database in the CLIPS environment offers a number of advantages. First, the problems of the distributed system immediately vanish. Concurrency updates involve only a single location where the data needs to be changed. In the distributed system, multiple occurrences of the data require multiple updates. Historical data must reside in the CLIPS environment where the knowledge base modules can directly access without going through an intermediary communication module residing in HyperCard, who must collect it from a real-time data stream and the various CLIPS modules of [PI]. Because historical data is readily accessible, the user is no longer prompted for information that already exists in the database. With this approach, the expert system needs only one set of data access mechanisms, instead of four. Second, file management considerations are handled by a single piece of software, thus avoiding the possibility of file lock-out which can occur when multiple pieces of software want to access the same file. [8] Third, the knowledge bases no longer need to worry about opening and closing files. Accesses to historical data involve fewer rules, making it easier to retrieve and store information. Fourth, functions built into the relational model, such as record insertion, selection, updating, and deletion allow off-line data manipulation to be performed once the expert system returned to ground, something that is not offered by the distributed set-up. The real-time considerations are also important for this solution. As will be seen later, this implementation scheme provides access times that fall within the real-time definition for the [PI] expert system.

CHAPTER 2
EMBEDDING A CUSTOMIZED REAL-TIME DATABASE WITHIN AN EXPERT SYSTEM

2.1 Introduction

A number of considerations must be made when building a database for an expert system, especially when the database is to operate in real-time. Many of these considerations are application dependent. For example, selection of the host language in which the database will be written is for the most part dependent on the representation scheme already selected for the expert system. Likewise, the way the data is stored and retrieved should be a direct procedure involving little or no interfacing to an external environment. The more interfacing required, the more overhead will be present in the final product. This overhead can result in a waste of memory and access time. Closely related is the way the data is to be inserted into the knowledge base. The database needs to be compatible with the expert system language and paradigm, else extra overhead can creep into the system. Finally, the real-time behavior must fall within the bounds of the time constraints. This is dependent on the definition of real-time imposed by the external environment, which is in itself related to the system's application.

2.2 The Host Language

Considerations for the host language include selection of the method for manipulating the database. One method, which will be described first, involves an application program with embedded capabilities to access an external database. This architecture uses procedure calls and a temporary storage pool for information retrieved from the database.

Because it must perform a variety of ordinary computational tasks, it is usually necessary for the application program to do more than just manipulate the database. For example, a program used by an airline to book reservations does not only need to retrieve from the database the current number of available seats on the flight and to update that number. It needs to make a decision: are there enough seats available? It might well print the ticket, and it might engage in a dialog with the user, such as asking for the passenger's frequent flier number. [16]

Thus, this approach involves writing the database in a conventional host programming language such as C or even COBOL. The host language is used for making decisions, for displaying questions, and for reading answers. In fact, it is used for everything but the actual querying and modification of the database. [16]

Here, the commands of the data manipulation language are invoked by the host language program in one of two ways, depending on the characteristic of the DBMS. (1) The commands of the data manipulation language are invoked by host language calls on procedures provided by the DBMS. (2) The commands are statements in a language that is an extension of the host language. Possibly there is a preprocessor that handles the data manipulation statements, or a compiler may handle both host and data manipulation language statements. The commands of the data manipulation language will thereby be converted into calls to procedures provided by the DBMS, so the distinction between (1) and (2) is not a great one. [16]

In this configuration, there is local data belonging to the application program, data that is manipulated by the program in the ordinary way. Embedded within the application program are procedure calls that access the database. A query asking for data causes the answer to be copied from the database to variables in the local data area. If there is more than one answer to be retrieved when a fetching procedure is called by the application program, then these solutions are retrieved one at a time. [16]

The disadvantages with this approach are obvious, especially for real-time applications. The host language does not do the actual data manipulation. It relies on procedure calls, or messengers, to do its job. The application program is not directly tied to the database. It uses a local pool of information through which data is shuttled to and from the database. The local area acts as an intermediary liaison that sits between the application program and its database. In real-time expert systems, the waste in time and overhead used to send procedure calls and temporarily store information may be unacceptable.

An alternative approach is to bring the database so intimately close to the expert system that they become almost one and the same. Here, the host language is the same language in which the application program is built. By writing the database in the expert system's host language, the procedure calls are no longer present. Manipulation of the data in the knowledge base is now done through the same rules that were used in the application. Because the database is an integral part of the knowledge base, the expert system can perform reasoning based directly on the database, which may reside in the knowledge base's list of facts. Using the airline example, the expert system can bind the frequent flier number contained in the database directly into one of its rules and decide whether or not the passenger deserves a discount.

With this approach, manipulation of data is done through commands which are extensions of the expert system's rules. For example, data insertion involves triggering the rules that perform the insertion. Once certain conditions are satisfied, such as the fact that an insertion has been requested by the application program, and the fact that the record to be inserted contains the passenger's name and frequent flier number, several rules are fired to insert the information into the database. Likewise, data retrieval involves triggering the rules which perform the retrieval procedure. By specifying the fact that a retrieval has been requested, that the number of miles the passenger has traveled is the parameter to be retrieved, and that the passenger's name and frequent flier number are Joe and 123456, several rules are fired to retrieve the requested information. Once the information has been retrieved, the expert system can reason directly with the data to see if the passenger deserves a discount. In other words, the rules perform the DBMS functions.

Because the database resides within the expert system's knowledge base, there is no local data area where information is temporarily stored. The DBMS functions in the application access the database directly without using procedure calls written in C or COBOL. The DBMS functions can retrieve, store, update, and delete information as needed because the data it manipulates sits directly in the expert system's knowledge base. This is definitely the better approach, especially for real-time systems with a small database.

The disadvantage with this approach is that it may actually become too unwieldy for systems with large databases. The rules themselves may not be proficient at searching through an extensive array of information. Furthermore, the expert system may impose memory constraints which would make storing an extensive database within its knowledge base impractical. The limits of this methodology are therefore mostly application dependent. For [PI], a system with relatively small memory requirements for its data, this approach is a viable solution.

2.3 Knowledge Base Considerations

It should be noted that, at this point, discussion will proceed assuming that the host language selected for the database is the same as that used in developing the expert system, which would be the more viable approach for a real-time expert system with a small data set. With the host language considerations already made, the appropriate framework, by which the database is to be embedded within the knowledge base, should be pursued.

Here, much of the consideration is centered around the expert system shell already chosen for the application. It is the shell that determines the appropriate compatible paradigm to be used for the database. In general, expert system shells allow knowledge to be represented in one of two ways: framed-based reasoning or rule-based representation. The types of knowledge that can suitably be structured using a frame-based organization can range from collections of related facts, to relationships between such collections, to rule-based and even procedural representations of knowledge. In rule-based tools, knowledge is conceptually represented as IF/THEN statements as described before. In some sense rule-based reasoning is a subset of frame-based reasoning. What follows will first be a description of the framed-based representation paradigm, then a description of the rule-based paradigm. [17]

The knowledge structures provided by a frame-based form of representation facilitate the development of a knowledge-based application in two ways. First, they assist the knowledge crafter in understanding the relationships that exist among the data being assembled. Second, they enhance the ability of the inference engine to operate on those data as well as on those structural relationships.

A frame can be viewed as a collection of related information about a topic. This information may be factual or procedural. A frame may be taken to represent a class of similar objects; other frames, representing subclasses or specific instances of those objects, can be formed from the initial class frame. The properties contained in a class frame can be inherited by its subclass and instance frames. Rule-based and procedural knowledge representations can operate on frame-based representations. A style of programming, termed object-oriented programming, has been developed on the frame foundation.

Frame-based representations offer an attractive way to structure a knowledge-based application and, together with frame-based reasoning, offer a powerful way to analyze problems. They also facilitate the rapid prototyping of knowledge-based applications and the modification of the knowledge base.

One of the major dissatisfactions expressed with frame-based reasoning concerns efficiency. The structures and capabilities of this representation scheme offer a wide variety of benefits, but these benefits are achieved only at a price. The dynamic ability to modify the knowledge structure or to modify the facets associated with a slot of a frame during execution of the application system requires that a considerable amount of checking be performed at execution time. The computer cannot store a fact in a frame with the execution of a single store command. Instead, the various facets governing the slot must be referenced to verify that storage of this particular value is permitted, to check whether this action is to be trapped, and so on. A simple store instruction that would involve only a single computer command in systems using other forms of knowledge representation can easily explode into tens of hundreds of instructions in a frame-based scheme. This expansion, of course, is accompanied by a degradation in the application's execution speed. [17]

In rule-oriented expert system shells, knowledge representation is based on the logical statement IF <predicate> THEN <consequent>. Using such statements, knowledge crafters formulate the knowledge they obtain form the experts into sets of such rules. The inference engine then analyzes and processes these IF/THEN rules in one of two ways, backward or forward. In backward-chaining, the inference engine works backward from hypothesized consequents to locate known predicates that would provide support. In forward-chaining, the inference engine works forward from known predicates to derive as many consequents as possible.

One of the most distinguishing features of rule-based developments tools is the pattern matching capabilities of the inference engine. This allows them to match a new data value with only those rules that reference that data value. Thus, predicates need only be tested only for those rules that might be affected by the new value. The effectiveness of the pattern matcher in an inference engine can significantly affect the tool's execution efficiency, a critical consideration for real-time systems.

Another characteristic of some pattern-matching algorithms is the ability to evaluate a given predicate only once, even though it might appear in more than one rule. Not only do such pattern-matching algorithms contribute to the efficiency of the inference engine's operation, but they also prevent some of the side-effect problems that can occur from multiple evaluations of a predicate. Some inference engines are much less sophisticated and much more brute force in approach. Rather than working only with those rules whose predicates include new facts or newly provided data items, some forward-chaining inference engines arbitrarily test every rule in the knowledge base to see if it can be fired. If any rules are actually fired, then every rule in the knowledge base will again be tested, and so on until no further rules can be fired. Obviously, the inference engine of such tools are inefficient.

Therefore, when building a customized real-time database in an expert system shell, the considerations to be made are largely centered around the representation scheme on which the shell is based. The paradigm may be frame-based to take advantage of the powerful structural relationships offered by the frame-based representation scheme. Or, it may be some other paradigm, more suited toward taking advantage of the pattern-matching capabilities of rule-based inference engines.

Furthermore, efficiency is an important factor for real-time expert systems. The large overhead imposed by the framed-based scheme needed in providing the storage capabilities of a DBMS may be impractical for real-time applications. On the other hand, the pattern-matching facilities of a rule-based inference engine may be so brute force that a pattern-matching paradigm will cause poor execution performance when storing, retrieving, updating, and deleting information from the database.

2.4 Data Model Considerations

The next major consideration is the database itself. To start with, a data model, formally defined as a notation for describing data and a set of operations used to manipulate that data, must be chosen. There are many data models that can be considered, including the entity-relationship model, relational data model, network data model, hierarchical data model, and object-oriented model, each with their own special capabilities, characteristics, and functionality. [16] Although data models have a rather abstract characterization of how data are to be manipulated, the following provides some distinctions that should be noted:

Purpose. In general, models serve the purpose of providing a notation for data and notation on which a data manipulation language is based. The entity-relationship model, on the contrary, provides a notation for conceptual design, before the implementation of the model of whatever DBMS is finally used. It therefore lacks a definition of how the data is to be manipulated. In some sense, it is not really a data model.

Object- or Value-Orientedness. The network, hierarchical, and object-oriented models come with object identity and are therefore called object-oriented. Value-oriented models, such as the relational and logical models, are declarative. That is, they support query languages. Nondeclarative models have been known to require less optimization.

Dealing with Redundancy. All models provide some way of helping the user prevent data from being stored more than once. Redundancy tends to waste memory space and requires concurrency control to keep the data consistent. Object-oriented models are able to cope with redundancy better than the other models because they allow the user to create a single object and refer to it using pointers from many different locations.

Dealing with Many-Many Relationships. Frequently, a database needs the capability to store many-many relationships, where each element in a group is related to many of another group and vise versa. For example, the relationship between students and courses is that each class contains many students and each student takes many courses. Designing a storage structure to allow the querying of these many-many relationships is not easy. Each model deals with this problem differently. The relational model puts the problem at the design level, while the network model does not allow many-many relationships at all, forcing the designer to factor the problem differently.

Whatever data model is finally chosen for implementing the DBMS in the expert system shell, however, the functionality must as a minimum include the abilities to (1) store data, (2) retrieve data, (3) change data, and (4) remove data. There may also be indexing functions to reduce data search time, ability for storage of views, security capabilities for data confidentiality, integrity checking mechanisms like masking and placing data range constraints, and locking to prevent the lost update problem in shared systems. [7] Nevertheless, the least a DBMS must be capable of doing is storing, retrieving, changing, and removing data. Taking the relational model as an example, the notation that may be provided for these data access functions might include INSERT, SELECT, UPDATE, DELETE (in SQL); I., P., U., D. (in QBE); INSERT, LIST, CHANGE, DELETE (in NOMAD); APPEND, DISPLAY, REPLACE, DELETE (in dBase). Although the notation may vary, the functionality is the same.

2.5 Real-Time Constraints

For real-time expert systems, there is one more important consideration. This concerns the real-time constraints in which the database must behave. Again, the definition of real-time behavior is largely dependent on the application for which the expert system was developed. The constraint may come from the external environment in which the system operates. Or, it may come from the expert system itself, as when information needs to be accessed before a specified deadline in order to perform acceptably. For the American Express Authorizer's Assistant system, a seventy second time limit is the constraint. For [PI], the DBMS functions must execute sufficiently fast to prevent the database from noticeably holding up the rest of the system.

2.5.1 Definition of Real-Time

Many ideas of real-time exist. In some cases, real-time is used to signify "fast" or "faster". A system which processes data quickly is often considered to be a real-time system. An alternative idea of a real-time system is one which yields a response after a "small" number of processing cycles have passed. Given more processing cycles to work with, the system refines its response. Yet, a third concept of real-time can be given. A system is said to exhibit real-time behavior if its response time is of the same order as the time scale in which external events occur. This definition suggests a notion of real-time which concentrates on the relationship between response time and the length of time it takes for environmental events to occur. [8]

There is, however, a crucial distinction between real-time behavior that provides guaranteed deadlines and behavior that exhibits high speed. In practice, the two behaviors are linked. In general, guaranteed deadlines require manipulation of in-memory objects.

High speed refers not only to hardware enhancement, but also to main-memory techniques, such as caching, prefetching, and parallel access. Because the issues of real-time systems have been the subject of much discussion in the literature, some of the issues pertinent to building a real-time database will be presented.

2.5.2 Issues of Real-Time

One of the most important measures of performance in real-time systems is response time. In these systems, the application task is often broken up into sub-tasks called modules, much like the way [PI] is composed of different software modules, each with a different functionality. In some cases, groups of one or more modules are assigned to different processors for processing. Response time is dependent on factors such as interprocessor communications, allocation of resources for the modules, data storage scheme, priority of the processors, and concurrency control procedures used in regulating shared resources. [4]

Concurrent operations and contention over resources in such systems are usually modelled using queueing networks. The processors are represented as servers, and modules are thought of as customers. The rate at which modules are invoked is thus the rate the customers arrive. Queueing networks, however, are limited in that they can represent only some forms of priority relationship and synchronization between modules. Moreover, the complexity of the model grows quickly when more modules are added to the application. As a result, it is sometimes impractical to use the queueing model for judging response time. [4]

Fortunately, there is a better model, developed at UCLA, for estimating the response time in multi-modular system. This model takes into account the queueing delay of each processor and then sums the module response times according to the module priority relationships. As it turns out, the model serves to predict response times fairly well. [4]

Although more related to schedulers than to databases, one interesting issue under discussion in the literature concerns the case of multiple deadlines. A difficult problem dealt with in the design of real-time systems is the handling of sporadic tasks, which are tasks that come with hard deadlines and at random times. Because the arrival times of sporadic tasks are unpredictable in nature, it is difficult to design a system able to guarantee that the deadlines of the all the sporadic tasks will be met. One way to handle this problem is by having an on-line scheduler decide if, given the execution times and deadlines of new tasks arriving at the system, all the deadlines can be met. If they can, the system executes the schedule constructed by the on-line scheduler. If they cannot, the system tries to meet the deadlines of the most important tasks and allows the deadlines of the least important tasks to be missed. An on-line scheduler is said to be optimal if it is able to schedule all the tasks such that they can be feasibly executed by the system's processors. It is interesting to note that no optimal on-line scheduler exists if the tasks have a multiple number of deadlines. [14] This may be something to consider when building a real-time database with an on-line scheduler to handle tasks that are inherently sporadic in nature.

In some real-time systems, it may be appropriate to implement a scheduler according to a time-driven scheduling model. In this model, the importance or value of a task is a direct function of its execution time. This function may not always be linear. The goal of a time-driven scheduler is to decide which are the most important tasks to complete, and as a result maximize the overall value. Depending on the application of the expert system, this approach can work very well. On uniprocessor systems, however, this approach is known to impose high processing overheads that make it rather impractical. Under heavy loads, a time-driven scheduler has been shown to spend 80% of its time deciding which task to execute next, and only 2% of its time doing actual scheduling. Therefore, this approach can work advantageously or poorly, depending on the type of system used for the application. [18]

One more issue worth mentioning is the distinction between hard and soft deadlines. In some applications, it is imperative that time bounds are met. Examples of such applications include flight control systems, nuclear reactor processing systems, and many real-time control operations in industry. For such systems, the time bounds impose hard deadlines, else failure to meet the deadlines will result in some form of damage, like loss of lives, nuclear melt-down, etc. The other type of deadline is the soft deadline. This is usually expressed by some percent probability that a requested task will be carried out in a specified time. Soft deadlines are important to systems that have a bound on the response time, but failure to meet them does not result in death, disaster, etc. Examples in which such deadlines exist include telephone switching networks and computer networks. When building a real-time database, the type of deadline applicable to the system should be considered. [12]

2.5.3 Common Real-Time Mechanisms

There are several real-time mechanisms that can be employed when implementing a database. Some mechanisms focus on the process of database updating while others on query computation. Real-time behavior can be approached by making database updates and query computation speedier, or by making the database more intelligent. This may include both software and hardware enhancements. The mechanisms considered here will strictly be software-based solutions. They include query optimization, minimization of the number of database queries, permitting processes to share memory with the database, prioritization of updates, and establishing an update sampling frequency. [8]

Query Optimization. The objective of query optimization, as often discussed in the literature, is to take a query expression, find functionally equivalent expressions that require less evaluation time, and evaluate the expression that costs the least in terms of computation. This may prove useful in expert systems that use only a finite set of queries. Although flexibility is sacrificed for speed, much time can be saved if frequently used expressions are optimized in such a way that they do not need complete re-evaluation.

Minimization of Database Queries. Because all queries require some amount of time to be executed by the system, reducing the number of queries that are made to the database means reducing the total time spent processing database functions. With fewer queries, computational resources can be shifted away from the database and be used by other parts of an expert system, like process control or data quality monitoring. There are a number of ways of reducing the number of queries. They include the use of triggers, which allow data to be directly sent from the database to a display screen without actually performing a database retrieval operation, and the use of prefetching to keep data manipulation within the main memory as much as possible and to minimize the number of file accesses.

Shared Database Memory. In practice, databases tend to prevent their data from being accessed directly. Accesses are permitted only through some interface to avoid storage of invalid data and unauthorized data accesses. This kind of control, however, can impede the data access to a point that is unacceptable in a real-time environment. Accesses can be performed more quickly if processes are allowed to share memory with the database, for example by putting some or all of the database's information in the expert system's knowledge base. The disadvantage of this approach is that some control over the data is lost, resulting in relaxation of concurrency control and thus, possibly, data integrity.

Prioritized Updates. Sometimes it may be necessary to prioritize updates, especially when the volume of updates is high. Similar to the time-driven scheduling model, the goal is to perform updates that are more important and queue other updates for later processing. This approach assumes that a response time and value is assigned to each update request. Again, the value is some linear or nonlinear function of the response time. The updates must be prioritized such that the sum of the values of the executed updates is maximized.

Update Sampling Frequency. In some real-time systems, the sensors collect information at a rate that is faster than the database is able to process the information. A solution to this problem is to be selective about the information to be processed, or by setting a maximum update frequency. If done correctly, the database has enough time to digest one set of values and to update its views before the arrival of the next set of values. The drawback with this approach is that unprocessed values are lost if the system's data acquisition buffer is too small to temporarily store all the incoming values.

Of these real-time mechanisms, the most relevant to [PI] are those which speed up query computation, allow processes and results to be shared between modules, and eliminate as much file access as possible. The most frequently used queries should be identified and optimized. Sharing data between modules will eliminate much of the overhead involved when going through an intermediary communication module. Furthermore, minimization of file access would reduce reading and writing information to and from the disk, which are time-consuming processes that should be avoided as much as possible. Although prioritizing updates would be useful, the volume of data is not high. Therefore time-driven scheduling would have limited applicability. Similarly, because the data set is relatively small for [PI], setting an update sampling frequency is not needed.

CHAPTER 3
IMPLEMENTATION OF REAL-TIME CLIPSBASE IN [PI]

3.1 Introduction

The project on which many of the previous considerations are centered, [PI], is a real-time expert system developed using a Macintosh application called HyperCard and a rule-based knowledge shell called CLIPS. [1,5] Lacking an internal database and hooks to external database applications, CLIPS was provided with several custom-built DBMS functions to allow storage, retrieval, modification, and removal of real-time information. These DBMS functions were written within the CLIPS shell itself, hence CLIPSBase.

This chapter first describes the space experiment, Dome, for which [PI] was developed, the organization of the real-time data used in the system, and the data model chosen for the database. Then, it discusses the methods used to embed CLIPSBase into the [PI] expert system by giving a background on CLIPS, pointing out various host language constraints, identifying the knowledge base scheme, and detailing the DBMS functions implemented. At the end, the real-time mechanisms employed in the database are presented. These include query optimization, allowing results to be shared between different modules, and minimizing file access.

3.2 [PI] and the Dome Experiment

[PI] is currently tailored to meet the needs of the M.I.T. Dome Experiment. This specific experiment was chosen to illustrate the potential of [PI] in helping the Principal Investigator (PI) and, ultimately, in improving space experimentation in general. By doing so, the project team hopes to uncover some of the generic issues surrounding the use of expert systems in space science. At this point, the Dome Experiment will be described, followed by details of the [PI] system.

3.2.1 The Dome Experiment

Professor Laurence Young, at M.I.T., has implemented a series of experiments whose goal is to understand how the human body adapts to microgravity encountered by astronauts in space. As the Principal Investigator, he has designed the "Rotating Dome Experiment" in his motivation to understand the relationship between human adaptation to space and sensory cues. In the following, a description of the experiment is provided by Professor Young: [21,22]

"The purpose of the Dome Experiment is to study the interaction of several spatial orientation senses during and following adaptation to weightlessness. Normally, all the senses (visual, vestibular, proprioceptive, tactile) act in harmony during voluntary head movements. In orbit, however, the otolith signals, acting as linear accelerometers, no longer produce signals which the brain can use to deduce the angular orientation of the head with respect to the vertical—and of course the vertical itself ceases to have any real significance. Nevertheless, the brain still searches for a reference system, within which it can place external (scene) and body position measurements. Visual cues, both static and dynamic, as well localized tactile cues, may become increasingly important in signaling spatial orientation as the brain adapts by reinterpreting otolith signals to represent linear acceleration, rather than tilt of the head with respect to the vertical. Semicircular canal cues, which normally signal head rotation, are not necessarily affected by weightlessness, but some evidence suggests that their influence also may be altered in space.

"Understanding of the level of brain adaptation to altered gravio-inertial forces may help to explain and possibly alleviate the symptoms of space motion sickness, which are thought to be related to sensory-motor conflict concerning spatial orientation.

"The hypothesis is that, in the course of exposure to weightlessness, visual, tactile, and proprioceptive cues will all become increasingly important relative to vestibular (particularly otolith) information in the judgement of body motion.

The procedure of the experiment is to have a subject stare into a hollow dome covered with multi-colored 1.9 cm dots at a density of approximately 800 per square meter. This dome rotates at various speeds and directions, while several measurements are made. The operation normally takes an hour with two astronauts, alternating as operator and subject. When the dome rotates, the subject, after some delay of several seconds, feels the illusion that he himself is rotating instead of the dome. The latency of the illusion's onset and its magnitude are recorded using a joystick manipulated by the subject. The data coming from the joystick is supposed to be of good quality. Upon inspection, it is relatively easy to interpret the amount of latency, magnitude, and other characteristics with which the PI is able to determine how adequate the test was, and if unexpected responses were encountered. Other measurements in the rotating dome experiment include video recording of the ocular torsion (rotation of the eyeball), strain gauge response resulting from the subject's neck torque, and electromyography (EMG) signals resulting from neck muscle activity.

"The first part of the operation is *unstow* and *setup* of the dome, TV cameras and recorder, and a portable oscilloscope. The next step is *subject preparation*, including the application of neck electromyography (EMG) electrodes, a contact lens and a bite-board.

"The experiment is paced by a dedicated computer, the Experiment Control and Data Systems (ECDS), which generates instructions, starts and stops the dome rotation according to preprogrammed sequences, acquires, digitizes and transmits data, and permits routing of analog test signals for hardware testing and for calibration.

"A brief *test* phase consists of verifying, on the oscilloscope, that each of the signals is coming through cleanly and with the correct polarity, and that the dome runs.

"A *calibration* phase consists of monitoring (and having the ECDS store) standard subject initiated movements of hand and head. The contact lens is irrigated to make it stick to the eye and the eye-camera is set and focused.

"Each *run* contains 6 *trials*, with the three possible dome speeds (30, 45, 60 degrees/second) and two directions (clockwise and counter-clockwise) arranged in a different fixed order for each of six possible *runs*. Each trial consists of a 20 second dome rotation at constant speed and a 10 second stationary period, so that each run consumes 3 minutes.

"Each subject will normally undergo three *conditions*. The *free float* condition has the subject restrained only by his or her bite-board and right hand on a joy stick. This is the basic dome experiment, testing simple visual-vestibular interaction. The otolith organs come into play in their failure to confirm head tilt, and the semicircular canals are relevant because of their failure to confirm any initial angular acceleration.

"The *neck twist* condition is like the previous one, except that the subject starts each dome trial by tilting his or her neck (which really means rotation of the rest of the body) in roll—always to the same side for each run. This condition is motivated by the hypothesis that proprioceptive signals from the neck lead to enhanced ocular torsion and perhaps also enhance neck righting reflexes.

"The *bungee* (or *tactile*) condition has the subject held down to a foot restraining grid plate (adjustable platform) by stretched elastic bungee cords. This condition, which places a localized tactile pressure cue under the feet, is to examine the substitution of tactile for inertial cues in weightlessness.

"Both for efficiency and to reduce order effects, the experiment usually is conducted in the above order for the first subject and in reverse order for the second, with the sequence of subjects kept the same during the flight.

"Following each subject's experience in the dome he or she is expected to report to the PI on Air-to-Ground to discuss qualitative sensations and any unusual occurrences.

"The final phase is *deactivation* and *stowing* of the equipment.

"During the course of an experiment seven types of data are recorded, as summarized below.

"*Identification* consists of the subject's ID (currently limited to 1-4 characters), and the dome run and trial.

"The *dome speed* and *direction* (TACHometer) is available as a series of pulses from a photocell located opposite silvered stripes on the back of the dome, and is computed as an alphanumeric value.

"The *joy stick* (JS) signal comes from a potentiometer adjusted by the subject. The subject uses it to indicate the strength of his or her visually induced rotation rate (not angle) relative to the speed of the dome. Full deflection of the potentiometer clockwise, for example, would indicate that the subject felt that he or she was rotating to the right (right ear down) and that the dome (which was actually turning counter-clockwise) was apparently stationary in space. It is a continuous signal, and it may be selected for display on the oscilloscope by the astronaut.

"The *biteboard* measures the *neck torque* by means of strain gauges attached to the support. It measures the tendency of a subject to straighten out his or her head to the upright when sensing that he or she is falling. It is principally sensitive to roll strain, but may respond to pitch and yaw torques as well. It is AC coupled with a 10 second time constant, so only changes in neck torque are recovered. It too can be selected by the astronaut.

"The *neck muscle* EMG from the right and left sides are also indicators of the initiation of righting reflexes to straighten the head. They normally consist of a low level of noise (both biological and instrument) during rest, and a burst of wide band activity during muscle contraction. We are interested primarily in the direction and timing of these bursts.

"The *ocular torsion* (OT) is measured by a video camera focused through a hole in the dome on the subject's right eye. Automatic data analysis of the OT is made possible by the opaque landmarks on the contact lens, which adheres to the eye briefly by application of distilled water. This measurement is very sensitive to camera adjustment, and the operator must assure proper focus, centering on the lens and bite-stick marks, and non slippage of the lens.

"The *neck angle* measures body sway, since the head is held stationary by the bite board. To accomplish this, a second video camera is aimed at the astronaut's back, suitably marked for automatic data reduction."

3.2.2 The [PI] System

Because of the problems mentioned before of limited resources in space, such as small bandwidth and lack of real-time access from the ground to data and astronauts, the [PI] project was initiated to take advantage of expert system technology. The goal of the project is to provide advice to the astronauts that the PI would provide if he or she were on board the Shuttle during the experiment. The system would embody the knowledge and know-how of the PI in order to make changes in the experimental protocol in response to unexpected events in the external environment. These events may include malfunctioning of a piece of experimental equipment, poor data quality, and "interesting" data.

The system must therefore be eminently real-time. While response time need not be instantaneous, the system must be able to keep up with the events during a one hour Dome session in which nominally six runs occur. Astronauts are under a great deal of pressure to complete many experiments according to a schedule. They do not have much time that can be devoted to fixing equipment failures and designing new protocols when things go wrong. The system must be able to provide useful advice as needed in the event troubleshooting or instant analysis is required. The variety of circumstances under which the astronaut must deviate from the pre-defined protocol include: [2]

The experiment is running late. This could, among other things, be due to a late start or delay in performing some of the steps of the experiment. Since the ending of the session is strictly enforced, some parts of the experiment may have to be eliminated.

There are equipment problems. A piece of equipment may have failed, possibly degrading the quality of the collected data by eliminating one of the data sources. A decision has to be made as to whether to continue the experiment with degraded data or to spend valuable session time trying to troubleshoot and fix the problem.

The subject's collected data may be "interesting" or it may be useless. When the data is found to be "interesting", it might be desirable to perform addition runs on that subject. On the other hand, when subject provides "erratic" data that is useless, it might be desirable to concentrate on the other subject.

3.2.2.1 The [PI] Architecture

The present version of [PI] consists of the following modules:

(1) The *Data Acquisition Module* (DAM) collects and reduces the raw data from the on-board experiment equipment.

(2) The *Data Quality Monitor* (DQM) ensures that the incoming data is reliable and error-free.

(3) The *Protocol Manager* (PM) helps keep the experiment on schedule by monitoring the experiment's progress and suggesting modifications to the protocol when necessary.

(4) The *Interesting Data Filter* (IDF) recognizes experimental data that is likely to be "interesting" to the PI, and helps the protocol manager suggest ways to pursue the interesting results.

(5) The *Diagnostic and Troubleshooting Module* (DTM) helps the astronaut isolate, diagnose, and correct problems in the experimental equipment.

(6) The *Experiment Suggester* (ES) uses input from the IDF to construct new experiments that investigate previous "interesting" results.

(7) The *Executive* moderates all inter-module communications, and ensures proper and timely allocation of system resources.

(8) The *Human Interface* (HI) allows the astronaut to interact with most of the modules.

The current architecture of [PI] is shown in Figure 1. [2] When raw time-series data arrives from the ECDS computer, it is relayed to the DAM and DQM on the Data Computer for statistical analysis and reduction. The parameters extracted from the raw data include means, maxima, and signal quality. The extracted parameters are then relayed, via a serial port connection, to the Executive on the AI Computer where it is stored in a local database residing in the HyperCard environment. Any communication between the other modules on the AI Computer as well as any information retrieval is done so through the Executive.

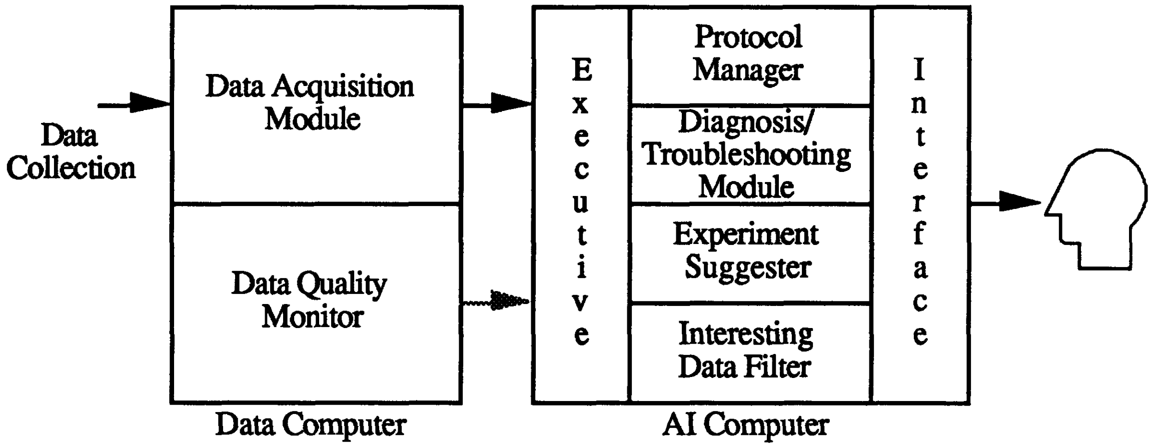


Figure 1. Architecture of [PI]

3.2.2.2 [PI] Data Model

Without getting into any detailed descriptions, the data from the experiment can be summarized as follows:

Trial Data. Trial data consists of (a) vection onset time, (b) number of dropouts, (c) average vection, (d) maximum vection, (e) first movement detected by the biteboard, (f) second movement detected by biteboard, (g) first movement detected by the EMGs, (h) second movement detected by the EMGs, (i) joystick quality, (j) joystick average, (k) biteboard quality, (l) biteboard average, (m) left EMG quality, (n) left EMG average, (o) right EMG quality, (p) right EMG average, and (q) quality override flag.

Run Data. Run data consists of (a) trial numbers (1-6), (b) trial quality, (c) start time, (d) end time, (e) duration, (f) environment, (g) condition, (h) subject, (i) mean vection onset time, (j) standard deviation (sd) of vection onset time, (k) mean number of dropouts, (l) sd of number of dropouts, (m) overall average vection, (n) sd of overall average vection, (o) mean maximum vection, (p) sd of maximum vection, (q) mean first movement detected by the biteboard, (r) sd of first movement detected by the biteboard, (s) mean second movement detected by the biteboard, (t) sd of second movement detected by the biteboard, (u) mean first movement detected by the EMGs, (v) sd of first movement detected by the EMGs, (w) mean second movement detected by the EMGs, (x) sd of second movement detected by the EMGs, (y) interestingness flag, and (z) level of interestingness.

Session Data. Session data consists of (a) run numbers (1-6) and (b) run quality.

General Information. There is general information used occasionally by the various modules. This includes (a) time left in the current session, (b) validity of various signals, and (c) troubleshooting time estimates for various possible malfunctions.

Subject Information. Subjects have attributes such as (a) name, (b) crew code, (c) MIT subject code, (d) height, (e) weight, and (f) age.

Altogether, there are five entities, namely trial data, run data, session data, general information, and subject information. Some of them have only a few properties, while others have many. To find a data model that fits this sort of organization is fairly straightforward because the data is inherently relational. For example, the subject entity has several attributes, which includes astronaut's full name, his or her NASA crew code (like PS, MS, etc.), MIT subject code (M, N, O, P, Q, R, S, or T), and other properties like height, weight, and age. The subject entity is related to the run entity through the subject name. Although the frame-based model is also a possible approach, the relationships between the trial, run, session, general, and subject entities do not closely fit the classes and subclasses of object-oriented knowledge representation. For this reason and the fact that the frame-based approach imposes a great deal of overhead, required in implementing frames and slots and in referencing slots to verify that a storage instruction is permitted, the relational model was chosen.

There are other reasons why the relational model was chosen. Because [PI] is a real-time system, response time is critical to the design consideration. Response time is largely dependent on how efficiently the DBMS functions are implemented. The relational model requires relatively little overhead to create records and fields. The cost of defining the structure of a relational organization is small. Tables are compact and make efficient use of memory. Furthermore, relational algebra is succinct yet versatile. In a rule-based shell such as CLIPS, relational access mechanisms require relatively less overhead to implement compared to the frame-based approach.

There are a number of ways to organize the data. It may be organized according to the five entities just given using five tables. Or, it can be organized into four tables, with the entities being run data, session data, subject information, and general information. Here, everything is the same as above, except the trial data would be organized into records within the run table. The trial numbers can be the primary key within the run table and the various attributes of trial data can be organized into fields, each field being a property of the trial data. Likewise, the run numbers can be the primary key within the session table, and the various attributes of the run data can by organized into fields, each field being a property of the run data. Because the procedure for designing a database can get rather complicated, the details will not be given here. A good reference is C. J. Date's *Introduction to Database Systems*. [7]

3.3 Methods Used to Embed CLIPSBase within [PI]

With the data model established, the next task is the actual implementation. In order to do so, familiarity in the knowledge shell as well as its constraints is first required. Then, the design criteria must be outlined according to real-time constraints, interaction scheme between different modules, the data set size, and system resource availability. The method by which records are to be integrated with the knowledge base must be determined. Finally, a set of DBMS functions are developed according to the above considerations.

3.3.1 Background on CLIPS

For the reasoning part of [PI], CLIPS (C Language Integrated Production System), was used. [5] CLIPS, inspired by OPS5 and using an ART-like syntax, was developed by NASA specifically to provide high portability, low cost, and easy integration with external systems. [13] Knowledge representation is done through a forward chaining rule language based on the Rete algorithm. CLIPS is reasonably simple yet powerful. It includes tools for debugging and can handle customized extensions to its language. Best of all, it is free!

As presented in the *CLIPS Reference Manual*, "the primary method of representing knowledge in CLIPS is a rule. The developer of an expert system defines rules which describe how to solve a problem. CLIPS provides an inference engine which attempts to match the rules to the current state of the system and applies the actions. The current state is represented by a list of facts.

"Facts are the basic form of data in a CLIPS system. Each fact represents a piece of information which has been placed in the current list of facts, called the fact-list. Rules execute (or fire) based on the existence or non-existence of facts. A fact is constructed of several fields separated by spaces. Any number of fields may be stored in a fact, and the number of facts in the fact-list is limited only by the amount of memory in the computer. Facts may be asserted into the fact-list prior to starting execution and may be added (asserted) or removed (retracted) as the action of a rule firing."

For example, the rules of a simple knowledge base may contain:

```
(defrule define-computer-programmer
        (it does not eat quiche)
        (it wears thick glasses)
        (it thinks in binary)
=>
        (assert (it is a computer programmer)))
```

If, during the course of the knowledge base's execution, the three facts *(it does not eat quiche)*, *(it wears thick glasses)*, and *(it thinks in binary)* exist in the fact-list, the rule *define-computer-programmer* is fired and the fact *(it is a computer programmer)* is asserted into the fact-list.

Among many capabilities, CLIPS allows variables to be bound to values, read/write functions to manipulate text files, printing to the computer display screen, logical operation of a rule's predicate conditional list, and fast pattern matching between facts in the fact-list and rules in the knowledge base. All of these functions are useful in building the DBMS of a relational database in a rule-based environment.

3.3.2 Host Language Constraints

Despite its versatility and power, CLIPS imposes a number of limitations on the developer. Unlike Lisp, where there is no distinction between data types, CLIPS makes a distinction between single field variables and multi-field variables. These two types cannot be used interchangeably within the rules. Certain functions applicable to one type of variable cannot be used on the other. As a result, a developer is forced to use tricks to swap values between the data types.

CLIPS provides a very small set of file access functions. Manipulation of data in files is therefore extremely limited. CLIPS allows only sequential reading, writing, and appending of data in the ASCII format to text files. Pointers cannot be provided to directly access values within a record structure, a capability readily available in languages like C. Many of the abilities of a procedural language are not present. These include definition of arrays, assignment operations to variables, data type conversions, and random access to file structures. Getting around many of these limitations can be a frustrating experience.

3.3.3 CLIPSBase Design Criteria

It would be desirable to have a system with instantaneous response. Every process, however, takes some time to execute. Some of these processes include data transfer, reasoning, updating the display screen, and transfering control between modules. Although time limits for data processing have not been strictly defined, one consideration is the fact that data is transfered from the Dome Experiment to the expert system regularly in 30 second intervals. Therefore, all the data processing, reasoning, and any other system processes must be performed quickly enough to keep up with the incoming data stream. With so many functions being performed during a single 30 second interval, little time is left for data manipulation. A DBMS function must therefore consume a very small fraction of this period whenever the database is accessed. Although this is a soft deadline, an inefficient implementation of CLIPSBase can cause information to accumulate in the system's data buffer, which is undesirable.

Interaction between different modules is currently being handled by the Executive module, which resides in the HyperCard environment. This poses a problem because most of the [PI] modules reside in the CLIPS environment. Whenever data needs to be passed between two CLIPS-based modules, it must first be transfered from the CLIPS shell to the Executive's HyperCard application. Then, the data is transfered back to the CLIPS environment where the other module exists. The design of the database must eliminate this indirect exchange of information. Some pool of information sitting in the CLIPS shell, allowing a common storage area, would be more efficient than the current scheme. The database must therefore be designed so that information can be transfered between two CLIPS-based modules without leaving the CLIPS environment. Anything less would be inefficient.

Compared to other expert applications, like the ones used by Nippon Life and the British Pension Advisor, the data set size for [PI] is relatively small. In a single Dome session, there are six runs, each run containing six trials, each trial containing fewer than fifty parameters. Even with all the miscellaneous session information and all the sessions conducted over the course of a number of missions and several baseline data collection periods, the volume of information can easily fit into the memory of a Macintosh equipped with eight megabytes of RAM. This makes it possible to minimize a great deal of data swapping between the system's main memory and its files. In fact, with this data set, it is possible to keep all the relational tables in memory and do all the data manipulation without first swapping information between RAM and the text files.

One limitation of the Macintosh platform is its lack of true multi-tasking. This is one of the main reasons why data collection and reasoning is performed on separate processors. The system, however, does provide some background processing capabilities. This is done through a piece of system software called Multifinder. Currently, Multifinder is the primary vehicle by which control of the processor can be switched between the HyperCard application and CLIPS shell. When reasoning is required, HyperCard first relinquishes control of the processor and passes it to CLIPS where reasoning takes place. Upon completing execution of the rules, control is passed back to HyperCard. Another design consideration is the fact that the DBMS can only execute its functions when control is passed to CLIPS, where the rules for the database reside. Although Multifinder allows some background processing of some tasks in the system's event queue, like mouse control and disk access, the rules can only fire when CLIPS is in control of the processor. The current version of Multifinder does not allow CLIPS to be run as a background task.

3.3.4 Knowledge Base Scheme

The next task is to identify the method by which relational tables can be embedded into the knowledge base. Since CLIPS only allows data to be represented in the form of facts in the fact-list, this is the way the relational tables must be defined. In the relational model, there are a number of tables, each containing field names that describe their contents. The contents are structured according to records, which can vary in number and field volume. Sometimes, there is a primary key that allows every record to be distinct from other records within a table. For example, the following is a table designed according to the relational model:

Table 1: My Magazine Collection

TITLE	DATES	VOLUMES	MAIN FEATURE
MacTutor	10/89 to 9/90	V5N10 to V6N9	Mac Development
Presentation Products	12/89 to 12/90	N/A	Computer Products
Technology Review	1/88 to 12/90	N/A	Technology
Hustler	4/88 to 6/89	N/A	Interesting Photos
MacWorld	9/88 to 1/91	N/A	Macintosh

In this table, the primary key is TITLE because each magazine has a different name distinguishing it from the rest. The DATES, VOLUMES, and MAIN FEATURE fields are properties that modify the magazine entity.

The table would be implemented in the CLIPS fact-list by asserting the following facts:

(magazine field-name TITLE DATES VOLUMES MAIN-FEATURE)
(magazine field-width 30 20 25 20)
(magazine record 1 MacTutor 10/89-to-9/90 V5N10-to-V6N9 Mac-Development)
(magazine record 2 Presentation-Products 12/89-to-12/90 N/A Computer-Products)
(magazine record 3 Technology-Review 1/88-to-12/90 N/A Technology)
(magazine record 4 Hustler 4/88-to-6/89 N/A Interesting-Photos)
(magazine record 5 MacWorld 9/88-to-1/91 N/A Macintosh)

Here, each fact is prefixed by the table name (*magazine*), fact type (*field-name*, *field-width*, or *record*), and a record index (if it is a record). The table name allows multiple tables in the fact-list to be distinguished from each other. The fact type allows facts within a table to be distinguished between a field name (describing its contents), field width (defining the character length of the field), and record (containing the actual data). Within the records, the index allows DBMS functions to access records according their position in the table, in the event a primary key is not available to distinguish between records.

This type of structure is compact, efficient, and flexible enough to allow any number of tables with any number of records and fields. When a new record is inserted into the table, it is simply appended into the fact-list with an index of 6. When a new row is created to accommodate another field name, the facts are expanded by attaching a new field at the end of each fact. Record accesses are quick using the CLIPS pattern matching capabilities. To retrieve a value within a record, a predicate such as

(magazine record ?n MacWorld ?volumes ?dates ?feature)

will automatically bind the DATES field to the variable *?dates*, VOLUMES field to the variable *?volumes*, and MAIN FEATURE field to the variable *?feature*. This is much like the relational statement *"select VOLUMES DATES MAIN-FEATURE from magazine where TITLE equals MacWorld"*. No search is required because CLIPS does all the work through its fast pattern matching abilities. As a result, this makes CLIPSBase very efficient and flexible at record retrievals.

The most important feature of this approach, however, is the fact that the tables all reside in the CLIPS shell. This makes information storage and exchange between the different knowledge modules clean and efficient. By providing a common pool of memory in the form of tables in the fact-list, knowledge base modules like the Diagnosis/Troubleshooting Module, Protocol Manager, and Interesting Data Filter can share information directly through the CLIPSBase DBMS functions. This is remarkably advantageous over the distributed file system where the Executive had to handle any data storage and communication needs. Now, the modules can fulfill these needs through a common set of functions without resorting to an intermediary module. Furthermore, the Executive can devote its efforts to resource allocation and control over module execution priorities.

3.3.5 Implementation of Access Functions

With the record structure defined, the DBMS functions are implemented. Among other things, the basic features of a relational database should be supported. This includes table creation, retrieval of a portion of a table, linked selection among a multiple number of tables, record and field insertion, record and field deletion, and multi-parameter updates. Here, the fundamental functions of CLIPSBase are described to demonstrate the database's abilities. A common feature among these functions is that updates to the disk files are performed concurrent with the updates to the fact-list. This is done to prevent loss of data in the event of a system crash, which can sometimes occur unpredictably on the Macintosh.

3.3.5.1 Table Creation

Before a table can be used, it must be created. The *create table* command is provided for this purpose. The parameters it receives are the table name, the field names within the table, and the character widths of the fields. The command sets up a table both in the fact-list and in a text file, where the field names and their widths are specified. With the table format in memory, it is ready to accept records.

3.3.5.2 Data Insertion

Data is inserted using the *insert* command. If a record is to be inserted, the parameters it takes are the table name and the record's field values. The command automatically inserts the new record into the fact-list and appends the text file. If a field is to be inserted, the parameters it takes are the table name, the new field name, and its character width. The command expands the facts in the fact-list by attaching the new name to the list of field names, the new width to the list of field widths, and asterisks to the records as padding for unknown values. It also expands the records in the text file accordingly.

3.3.5.3 Data Update

Records are updated using the *update* command. The parameters it takes are the table name, a list of fields and their new values, and a list of conditions under which records are to be updated. The list of conditions describes the relational algebra applied to the field names. The command evaluates the relational algebra and determines which records need to be updated. Then, it modifies the records according to the list of field names and their new values. Both the records in the fact-list and the text file are modified.

3.3.5.4 Data Retrieval

Data is retrieved using the *select* command. The parameters it takes are the table name, a list of fields, and possibly a list of conditions under which records are to be selected. If a list of conditions is specified, it gets evaluated to see which records are retrieved. If no list is specified, then all the records in the table are retrieved. The list of fields contains the field names—which can be in any order—to be used in the selection process. A new table is then created in the fact-list containing the results of relational evaluation. The table consists of facts prefixed by asterisks to indicate that it is the result of a *select* command.

3.3.5.5 Data Deletion

Data is removed using the *delete* command. If records are to be deleted, it takes the table name and a conditional list as parameters. The conditional list is evaluated according to its relational algebra, and the corresponding records are removed from both the fact-list and text file. If a number of field rows are to be deleted, it takes the table name and a list of fields. Here, no relational algebra is evaluated. The specified field rows are simply removed from both the fact-list and text file.

3.3.5.6 Linking with the HyperCard Environment

Because CLIPS and HyperCard do not provide facilities to communicate with each other, a tool called HyperCLIPS was developed by Chih-Chao Lam at Stanford University to provide simple inter-application communication. HyperCLIPS permits developers to build user interfaces in HyperCard while issuing knowledge-intensive queries for processing by a CLIPS knowledge base. It consists of several HyperCard XCMDs that initialize the system, two Macintosh drivers that provide a channel of communication between the two environments, and extensions to the CLIPS knowledge shell that allow it to poll for commands coming from HyperCard. [15]

Commands can be sent from HyperCard to CLIPS by issuing a command in HyperCard using its HyperTalk scripting language, allowing CLIPS to process the command according to its rules, and receiving the results through the channel provided by extensions in the knowledge shell. With HyperCLIPS, queries can be thus sent directly to the database where they get evaluated, and where results can be returned directly to the HyperCard application. [15]

3.3.5.7 Examples of CLIPSBase in Use

A number of examples of how CLIPSBase functions are used in CLIPS will now be given. The details of their syntactical structure and functionality can be found in Appendix A: *CLIPSBase User's Guide*. The examples given here are purely for demonstration of how they can be embedded within the [PI] knowledge rules.

create table: setting up a subject table

```
(defrule set-up-subject-table
        (initialize the system)
=>
        (assert (create table Subject fields name crew-code MIT-subject-code height
                weight age widths 30 5 3 7 7 5)))
```

insert: inserting a new record

```
(defrule insert-trial-parameters
        (put new trial parameters into run table)
=>
        (assert (insert into Run values 1 bungee MS1 t41*32*20 76 34 45 55)))
```

select: retrieving a portion of a table

```
(defrule get-run-quality
        (get quality of run 4)
=>
        (assert (select quality from Session where run eq 4)))
```

update: modifying a table value

```
(defrule override-run-quality
        (astronaut says quality of run 4 is bad)
        ([pi] says quality of run 4 is good)
=>
        (assert (update Session set quality eq bad where run eq 4)))
```

delete: removing a record from a table

```
(defrule remove-astronaut-from-table
        (astronaut MS1 not on mission)
=>
        (assert (delete from Subject where crew-code eq MS1)))
```

where: defining relational algebra

```
(defrule get-bad-run-records
        (declare (salience 6100))
        (find records with bad runs in session table)
=>
        (assert (in Session where quality eq bad)))
```

insert file: inserting a record from a text file

```
(defrule insert-new-trial-values-from-file
        (get text file record)
=>
        (assert (insert into Run file New-trial)))
```

fast-select: retrieving a value using optimized selection

```
(defrule get-run-quality
        (get quality of run 4)
=>
        (assert (fast-select quality from Session where run eq 4)))
```

linked-select: retrieving information using relationships between tables

```
(defrule get-run-quality-of-MS1
        (get run quality of MS1 from session table)
=>
        (assert (linked-select quality from Session where
                subject eq
                fast-select name from Subject where crew-code eq MS1)))
```

display-table: displaying a table onto the screen

```
(defrule show-subject-table
        (show subject table)
=>
        (assert (display-table Subject)))
```

load-table: "smart-loading" a table into memory

```
(defrule load-general-information-table
        (prefetch general information table)
=>
        (assert (load-table General)))
```

do-load-table: "force-loading" a table without checking

```
(defrule load-table
        (reload run table)
=>
        (assert (do-load-table Run)))
```

save-table: saving a table in memory to its file

```
(defrule save-session-table
        (save session table)
=>
        (assert (save-table Session)))
```

retract-table: removing a select evaluation

```
(defrule evict-run-information
        (remove evaluation on run table from memory)
=>
        (assert (retract-table Run)))
```

remove-table: removing a table from memory

```
(defrule remove-run-table
        (remove run table)
=>
        (assert (remove-table Run)))
```

3.4 Real-Time Mechanisms Employed

Because time is indeed a precious commodity in [PI], efficiency of the DBMS functions is important. To that extent, real-time mechanisms are employed to minimize processing performed when manipulating data. These mechanisms include query optimization, allowing results to be shared between different modules, and minimizing the amount of file access.

3.4.1 Query Optimization

Query optimization presents both a challenge and an opportunity to relational systems: a challenge because it can become a necessity in some real-time applications, an opportunity because relational expressions are at a sufficiently high semantic level that optimization is feasible in the first place. The overall purpose of an optimizer is to choose an efficient strategy for evaluating a given relational expression. [7]

It should be noted that the definition of query optimization discussed here is not the same as that commonly held in database literature. In C. J. Date's *Introduction to Database Systems*, optimization is performed through a strategy chosen automatically by the system. In such a definition, it is the system, not the user, who decides what record-level operations are needed and in what sequence those operations are to be executed. The strategy typically includes (1) casting the query into some internal representation, (2) converting to canonical form, (3) choosing candidate low-level procedures, and (4) generating query plans and choosing the cheapest. [7]

Here, however, optimization refers to implementation of special purpose functions by identifying commonly used queries. The strategy of these functions is already set by the system and cannot be changed. This type of optimization is used by some real-time applications where the queries are predetermined and the processes of casting queries, converting queries, choosing procedures, and generating plans are not necessary. [8]

For [PI], the most heavily used type of query is the selection of values from the tables. An optimized command called *fast-select* was therefore provided. The *fast-select* command allows retrieval of a single value from the database by asserting the statement

(fast-select <field-name> from <table-name> where <field-name> eq <value>)

into the fact-list. Here, flexibility of the normal *select* command was sacrificed for speed. The parameters it takes are the field name, table name, and a restricted conditional. The conditional currently allows for the matching of a only one value to the field name because it works by retrieving the first value occurrence in a table. For example in the *magazine* table, a statement such as

(fast-select TITLE from magazine where VOLUMES eq N/A)

would result in the fact

(fast-select: Presentation-Products)

being asserted into the fact-list. Although an extended conditional list is not supported and only a single value is retrieved at a time, *fast-select* has an improvement in execution speed of about six times over the normal *select* function.

3.4.2 Sharing Results Between Different Modules

One of the virtues of having a fact-list is that evaluations of relational algebra can be shared among different modules. When a complicated *select* function is executed, the result is asserted into the fact-list and can be used by any number of modules without re-evaluation of the *select* statement. The fact-list acts as a common pool where results can be temporarily stored and communicated to the knowledge bases, much like the blackboard technique. [6]

Of course, the results of a *select* statement reside in the fact-list only until the next *select* statement is issued. The current implementation of CLIPSBase allows for only one set of results to be stored per table in memory. But even with one set stored per table, much time can be saved if its results are shared between different modules. The strategic design guideline is to issue *select* statements in a proper sequence such that modules can avoid re-evaluation in order to make use of the results.

3.4.3 Minimization of File Access

One of the most time consuming processes during the execution of CLIPSBase DBMS functions is access to text files. There are basically two ways in which DBMS functions can manipulate data. The first way is to swap tables in and out of the fact-list as needed. In this approach, only one table is stored in main memory at any time. Although this saves on memory usage, much time is spent removing a table from the fact-list and loading a new one in. The second way is simply to load all the tables into memory and avoid accessing their files as much as possible. The only access performed will be that of updating the records in the event of data insertion, updates, and deletion. This prefetching technique has been used by a number of applications to make database access more efficient. [3] The time savings using this method have been found to be a factor of two or three over the swapping scheme in CLIPSBase.

CHAPTER 4

ANALYSIS OF [PI] DATABASE IMPLEMENTATION SCHEME

4.1 Introduction

The implementation of CLIPSBase described in the last chapter is intended to enhance the functionality and/or performance of knowledge bases built in the CLIPS shell. The motivation behind the use of real-time mechanisms, the selected relational model structure, DBMS functions, and the data access scheme is to provide a means by which real-time applications can efficiently share and manipulate information. This chapter analyzes the CLIPSBase implementation scheme in terms of time and/or computation overhead required to perform various DBMS operations.

The database functionality has been to some extent tailored to the needs of [PI], as reflected in the design choices made for the real-time mechanisms employed. Therefore, many of the examples given will be extracted from a data structure suitable for use by the various [PI] modules. First, examination of the access mechanisms will be given in terms of the amount of overhead required to manipulate information in the database and how data organization can affect access efficiency. Second, discussion of the real-time mechanisms will entail the improvements attained through query optimization, shared processes and results, and file access minimization. Third, examples will be given of how DBMS functions can be used in the [PI] knowledge base. Fourth, discourse will cover the extendibility of CLIPSBase and how the reasoning capabilities can be built on top of the database functions. In its conclusion, the chapter gives a general assessment of the database implementation in terms of its functionality, including strengths and weaknesses.

4.2 Overhead Requirements

The overhead of CLIPSBase can be broken down into several components: the overhead required to interpret a relational statement, the overhead needed to evaluate relational algebra, the overhead constituting a search process, the overhead required to manipulate facts in the fact-list, and the overhead needed in defining the structure of the relational model.

Interpreting a Relational Statement. A CLIPSBase statement is issued by asserting it into the CLIPS fact-list. The statement would contain the actual SQL command and the required parameters for execution. In the form of a fact, a rule fires when the pattern corresponding between the sequence of fields in the fact matches the sequence of fields in the predicate of one of the database rules.

Usually, the fields of the statement are parsed by separating the list of conditions, specifying the relational algebra, from the other parameters like table name, field name, field width, etc. In the case of an *insert* command, the record is extracted from the rest of the statement, gets inserted into the fact-list, and gets appended to the table's corresponding text file. In the case of an *update* command, the new field values are extracted and used in the fact manipulation process. In all cases, the table name is extracted from the statement so that the rules know which table to operate on.

Evaluating Relational Algebra. Given a designated table, the next stage is the evaluation of the relational algebra contained in the list of conditions. The conditions are specified according a *where* clause embedded within the relational statement. The *where* clause consists of a sequence of and's, or's, field names, values, and comparators making up the tuple relational calculus. The clause would be evaluated in some order depending on the algebra used. For example, given a *where* clause such as *where A < 5 or B eq 3*, the conditional would be evaluated according to the order contained in *where (A < 5) or (B eq 3)*. The clause can be of any length and any complexity. The evaluation time, of course, would be greater for complicated, non-trivial clauses.

The tuples in the *where* clause specify the records on which the function is to operate. Once evaluation is completed, a fact containing a series of record indexes is asserted into the fact-list. The record indexes correspond to the assigned order for the records placed in the fact-list. The asserted fact looks like

(prefix-junk record-seq 1 2 3 4 5)

meaning that records 1, 2, 3, 4, and 5 have been chosen in the relational evaluation. The *prefix-junk* is a sequence of parameters used only by CLIPSBase during the processing of rules and has no significance to the developer. The overhead used in processing the *where* clause is probably the greatest compared to the other types of overhead. On a large table, over half the time required in executing a command can be taken up by this stage.

The Search Process. After evaluating the relational algebra, a search is performed on the records of a designated table. The search rules use the asserted record indexes that were asserted into the fact-list during the evaluation stage. The search process is relatively quick because it relies on the fast pattern matching abilities of the CLIPS inference engine to determine which records correspond to the record indexes in the asserted fact. The rules work by trying to bind variables in the rule predicate according to a statement like

(subject-table record 5 $?bunch-of-values)

This means that, given the table called *subject-table*, the rules must bind the variable *$?bunch-of-values* to the field values in record 5. Once the variable is bound, the values contained within it can be manipulated according to string functions, mathematical operations, etc. The overhead imposed by the search process, again, depends on the size of the table. Compared to the other stages, however, it is fairly insignificant.

Manipulating Facts. Manipulation of the facts can involve several different forms of operation, depending on whether the DBMS command requires data insertion, selection, updating, deletion, or display. When a new field is to be added in a data insertion command, the facts are manipulated by attaching the field name, field width, and asterisks to the appropriate facts, then removing the old facts, and finally asserting the extended facts. When data selection is required, values are extracted from the appropriate records, then reorganized according to the sequence of field names in the *select* command, then joined together into a string that contains the table name, fact type, and selected values, and finally re-asserted into the fact-list as a new table. When data updating is requested, the record fields containing the values in need of modification are spliced out of the facts, replaced by new values, and re-asserted into the fact-list. When a field needs to be removed from a table, the *delete* command uses a sequence of processes that is the opposite of the processes used when inserting a field. Instead of appending new values to a series of facts, it cuts them out. It does this by splicing a fact at the position where the field is located, removing the unwanted field, re-attaching the two fact segments, and re-asserting the fact into the fact-list. Finally, when a table needs to be displayed, the facts constituting the table are simply severed to remove the table name, fact type, and record index. The data that is displayed to the screen contains only the field names and their corresponding values.

Some of these processes require little computational overhead while others require a lot. Those that must do a great deal of field re-organization and fact splicing tend to take more time. Of the fact manipulation processes just mentioned, the data display function is the most time consuming. This is due to the slow screen update mechanism provided by CLIPS. As a result, the *display-table* function is used primarily for development purposes and not for the actual expert system application.

Defining the Relational Structure. Fortunately, the structures of relational tables are compact and make efficient use of memory. Little overhead is used to define the relational structure. The facts constituting a table contain only five elements: the table name, fact type, field names, field values, and the record index. The many-many relationship of the relational model is not implemented as part of the structure. Rather, the relationships are defined according to the rules that access tables using information from other tables. For example, the *linked-select* function is provided to allow selection of information from one table based on information from many other tables. The concepts of primary key and foreign key are left to the developer, who must make sure that a selection based on a many-many relationship is performed using proper entity types.

File Access. Another large piece of overhead is taken up by the file access mechanisms of CLIPS. This is due mostly to the fact that the CLIPS shell is not powerful at making file accesses, all of which must be performed sequentially. As a result, loading of large tables into the CLIPS fact-list is a slow process. This forces the developer in a real-time application to avoid reloading of tables and to do as much prefetching as possible. The time overhead of file access is mostly dependent on the way tables are set up in the knowledge base, as discussed below, and not on the overall size of the database. It is possible to have an extremely large database, provided that it fits into the computer's memory, and yet have efficient data access, simply by trying to manipulate data in small chunks.

4.3 Analysis of Data Organization Scheme

There are a number of ways in which relational tables can be set up in the CLIPS knowledge base. For example, an entity might be a *run*, which contains six *trials*, each trial with the properties *start time*, *end time*, *duration length*, *environment*, *condition*, *subject*, *joystick average*, *EMG average*, and *biteboard average*. The following are two ways in which a run table can be set up.

Table 2: Run Example 1

trial-#	start	end	length	envir	cond	subject	js-av	emg-av	bite-av
1	a1	a2	a3	a4	a5	a6	a7	a8	a9
2	b1	b2	b3	b4	b5	b6	b7	b8	b9
3	c1	c2	c3	c4	c5	c6	c7	c8	c9
4	d1	d2	d3	d4	d5	d6	d7	d8	d9
5	e1	e2	e3	e4	e5	e6	e7	e8	e9
6	f1	f2	f3	f4	f5	f6	f7	f8	f9

Table 3: Run Example 2

parameter	trial-1	trial-2	trial-3	trial-4	trial-5	trial-6
start	a1	b1	c1	d1	e1	f1
end	a2	b2	c2	d2	e2	f2
length	a3	b3	c3	d3	e3	f3
envir	a4	b4	c4	d4	e4	f4
cond	a5	b5	c5	d5	e5	f5
subject	a6	b6	c6	d6	e6	f6
js-av	a7	b7	c7	d7	e7	f7
emg-av	a8	b8	c8	d8	e8	f8
bite-av	a9	b9	c9	d9	e9	f9

Several tests were performed on various CLIPSBase DBMS functions using a Macintosh IIx to compare computation times required to operate on each table. Although both tables contain the same amount of information, there was a disparity in the results, as shown in Table 4. Here, results were obtained by averaging several computation times for each operation. All tests for each operation on a table were consistent within 1/10000 of a second. The command statements were constructed such that the same field values were retrieved from either table.

Table 4: Comparison of Overhead (in seconds) for Run Examples 1 and 2

DBMS Operation	Overhead for Example 1	Overhead for Example 2
fast-select	.0468	.0625
select	.2812	.4141
insert	.6015	.6328
update	1.1796	.9531
delete	.6015	.6875

The difference between the tables is that, although they contain identical information, Example 1 uses a fewer number of records but greater number of fields than Example 2. With the exception of the update operation, as indicated, CLIPSBase operations are more efficient at handling tables with a small number of records, though they may have a large number of fields, than tables with a large number of records, though they may have a small number of fields.

In terms of [PI], these are acceptable overhead values. During the course of a trial, it is estimated that a total of two *insert* instructions, ten *fast-select* requests, and perhaps two *update* commands will be issued among five or six tables, each table having an approximate data size of nine fields and six records. Altogether, the system would spend about four seconds executing DBMS functions during each trial. In the distributed file architecture, over fifteen seconds are spent in a single data retrieval operation just transfering data and passing processor control between the CLIPS and HyperCard environments. With the implementation of CLIPSBase this inefficiency is avoided, giving a time savings of at least eleven seconds per data retrieval operation! This is a remarkable improvement, especially since each trial is performed within a 30 second time constraint.

Another method for improving data organization is by breaking large tables into smaller ones. This is not the same as ignoring certain pieces of data. For example, the run table in Run Example 1 above can be broken down into two smaller tables called Run Assumptions and Run Calculations:

Table 5: Run Assumptions

trial-#	start	end	envir	cond	subject
1	a1	a2	a4	a5	a6
2	b1	b2	b4	b5	b6
3	c1	c2	c4	c5	c6
4	d1	d2	d4	d5	d6
5	e1	e2	e4	e5	e6
6	f1	f2	f4	f5	f6

Table 6: Run Calculations

trial-#	length	js-av	emg-av	bite-av
1	a3	a7	a8	a9
2	b3	b7	b8	b9
3	c3	c7	c8	c9
4	d3	d7	d8	d9
5	e3	e7	e8	e9
6	f3	f7	f8	f9

The idea is to group certain fields into a common category and putting the category's description in the table name. The effect of using smaller tables is that the overhead required in the fact manipulation process gets reduced because there are fewer fields within each record to deal with. Some time comparisons are given in Table 7:

Table 7: Comparison of Overhead (in seconds) for Run Example 1 and Run Calculations

DBMS Operation	Overhead for Example 1	Overhead for Calculations
fast-select	.0468	.0468
select	.2812	.2601
insert	.6015	.4375
update	1.1796	.7656
delete	.6015	.5781

The disadvantage of this method is that relational algebra is more complicated when accessing data across more than one table. This can, in fact, cause a greater overhead due to the increase in computation required in evaluating relationships between tables. In order for this approach to be effective, the database designer must predetermine how fields should be grouped in order to avoid inter-table data manipulation. For some expert systems, predetermination of the data accesses is possible. [PI] is one of these systems. For other systems, the gain in speed may not be enough to compensate for the loss in flexibility.

4.4 Analysis of Real-Time Mechanisms Employed

The real-time mechanisms used in CLIPSBase will now be discussed in terms of their effectiveness in reducing overhead. This overhead, as mentioned before, can come in different forms. Query optimization works by simplifying the computation required to evaluate a relational expression. Sharing processes and results is way of reducing the number of data access mechanisms needed in the system and preventing re-evaluation of relational statements that are frequently used by various modules within [PI]. More importantly, communicating data between different modules is achieved without going through an intermediary Executive module. File access minimization is achieved by prefetching tables into main memory and avoiding unnecessary delays in the middle of a real-time process.

4.4.1 Query Optimization

Query optimization is effective only in relative terms. It depends on how frequently certain database queries are made and how much time is saved by optimizing these queries. In the case if [PI], where at most ten data retrieval operations are estimated per trial, there would a reduction in access time of only a few seconds. This improvement will probably go unnoticed because it is relatively small compared to the savings achieved by preventing control and data from being passed between the CLIPS and HyperCard environments.

For example, when a query is made by the Diagnostic/Troubleshooting Module, the request must be sent to HyperCard, where it is interpreted and the desired information is collected from various HyperCard fields. This process alone takes several seconds. In CLIPSBase, the query is optimized in the sense that the information already resides in the knowledge base. With the distributed file system, data from the Data Computer is sent only to HyperCard fields. When information is needed, the data is first saved to individual text files belonging to various knowledge base modules, and then read by modules when they do reasoning. If after each trial, data coming from the Data Computer is, instead, sent directly from the serial port to CLIPSBase files, the retrieval process would not need to take the extra step of getting the data from the HyperCard fields. Here, optimization exists in the retrieval process. The computation is simplified because an extra step is avoided.

4.4.2 Sharing Processes and Results

Being able to share processes and information is probably the greatest advantage offered by CLIPSBase. Instead of using a multitudinous number of access mechanisms between different [PI] modules, the DBMS is reduced to a single set. The database establishes a standard in the form of SQL statements, which is more versatile and organized than the current access mechanisms offered by the distributed file system. This set is shared and it can be used to access a common pool of data.

The data resides in the knowledge base in the form of relational tables, allowing information to be shared between different modules. Relational evaluations are stored not in HyperCard fields which are not directly accessible to the reasoning modules, but within the fact-list where knowledge rules can simply bind the information into their predicates. The rules can be IDF rules, DTM rules, or PM rules. It makes no difference, because the data is consolidated into a centralized storage area and not divided into separate files as in the distributed file architecture. As a result, an evaluation performed by one module can be openly shared with many other modules, something that is not easily achieved in a distributed set-up.

There is yet another plus of having shared information. When a module needs to do reasoning based on historical data, all the necessary information will be available within the knowledge base. This allows the expert system to reduce the number of queries it must make to the astronaut. In the distributed file system, the historical information is hopelessly scattered in an unorganized format. As a result, a module is ignorant of what resides in the localized database of another module. This forces it to pose an unnecessary number of questions to the user, questions that can already be answered by information sitting in one of the files. In CLIPSBase, this ignorance is eliminated altogether, making it possible for [PI] to pose queries to which it definitely does not have an answer and reduce the amount of attention that the astronaut must provide to the system.

4.4.3 File Access Minimization

With the current implementation of CLIPSBase, file access minimization is achieved by keeping all the tables within the fact-list, and avoiding the information from being swapped between main memory and disk files. This is possible because the data size of [PI] is relatively small compared to expert systems like Nippon Life Underwriter's Aid and Britain's Pension Advisor, where the databases for these applications are much too large to fit into the memory of a microcomputer.

Swapping tables in and out of memory is an expensive disk process that should be avoided as much as possible. For [PI], this means that static data should be prefetched into memory and kept there so that information is readily available when modules need to do reasoning based on that data. Static data can include the subject and general information tables, containing astronaut names, crew codes, MIT codes, and other information that never change during the course of a session.

Also, accesses to the tables should be consolidated as much as possible. For example, instead of updating values within a table one at a time, the developer can take advantage of the *update* function's ability to modify a multiple number of fields within a record in a single sweep. CLIPSBase can then change the values in a table file all at once, which is more efficient then changing them individually.

4.5 Database Extendibility

One of the virtues of embedding a database in a knowledge shell is that reasoning functions can be built directly on top of the data model used in the application. Consequently, it is easy to make the database smarter. For example, sophisticated [PI] rules can be written based on the information in the session table to determine if a run is considered good or bad. The results of this analysis can then be stored in a historical analysis table where statistical rules can be applied to suggest a trend in the quality of the data. This new functionality would become part of the database, where any module can use it. Eventually, the functionalities of the database and reasoning modules would in a sense merge so that there would be really no distinction between issuing a DBMS command that performs data manipulation and firing DTM (or IDF or PM) rules that do reasoning based on data. Admittedly, this is a bold statement. It is, however, the trend in future architectures for data processing systems, where "knowledge and data interact to create information." [19]

4.6 Overall Assessment of Database Implementation

CLIPSBase comes with various types of overhead. This includes the overhead used in interpreting a relational statement, evaluating relational algebra, searching for records, manipulating facts, defining the relational structure, and file access. Of these, evaluating relational algebra and file access require the most time to execute. This is because the current implementation of CLIPSBase is not particularly fast at evaluating tuples and the CLIPS environment is very limited in terms of its ability to manipulate data in text files. This imposes some constraints on the developer to be careful about defining complicated relational statements and being efficient with certain data access functions. Despite these constraints, it is estimated that CLIPSBase can save at least eleven seconds worth of time in each retrieval operation, simply by organizing the data in the form of relational tables in the fact-list.

Efficiency of data access is to some extent dependent on how the tables are set up in the knowledge base. It is best to keep the number of records as small as possible. Certain CLIPSBase functions, like *fast-select*, *select*, and *insert*, are more efficient at manipulating a small number of records with a large number of fields, than at dealing with a large number of records, even if they have a small number of fields. Certain functions, like *update*, *delete*, and again *insert* are particularly sensitive to the number of fields in a record. As a result, tables should be broken down into smaller tables whenever possible. The method achieves some speed at the cost of some flexibility. Depending on the application, this can be either advantageous or disadvantageous. For systems like [PI], where much of the data accessing can be predetermined, the fields within a table should be grouped so that the DBMS functions have smaller facts to deal with. At the same time, however, there should be no need to access data across different tables.

The real-time mechanisms employed are query optimization, sharing processes and results, and file access minimization. Query optimization is effective only in relative terms. The optimization is achieved in [PI] because computation is simplified in the retrieval process where the step of collective data from HyperCard stacks can be eliminated. Sharing processes and results is the greatest advantage offered by CLIPSBase. One of the most wasteful processes in the distributed file architecture is its need to collect and transmit information through an intermediary Executive module. This has the effect of forcing modules to repeat many processes and pose an unnecessary number of questions to the astronaut. By centralizing the data in the form of tables and providing a common set of access mechanisms, there is greater flexibility and organization in the way data is manipulated. Furthermore, while file access is not one of CLIPS' strong points, it can be minimized to reduce delays in the middle of a real-time process. This is done by prefetching tables into the computer's main memory and preventing data from being swapped between the fact-list and text files.

Finally, embedding a database in a knowledge shell is a step toward the future architecture where knowledge and data work synergistically to produce information. This approach provides the ability to build functions directly on a given data model, adding functionality to the DBMS and making the database smarter. The vision is to have one day a database that is one and the same as the reasoning modules of [PI].

CHAPTER 5
CONCLUSIONS AND FUTURE RESEARCH

5.1 Conclusions

This thesis shows that embedding DBMS functions in the shell of a real-time knowledge application can reduce overhead required to manipulate data. In particular, the real-time application considered here is [PI], an expert system used to advise astronauts in space life science experiments. The advantages to be gained through this approach include elimination of communication time between different environments, reduced number of queries to the user, and decrease in the amount of reprocessing that is performed between modules. The real-time mechanisms that were implemented as part of the database architecture include query optimization, allowing processes and results to be shared between different modules, and minimizing the amount of file access. The ideas presented here are especially applicable to those applications that come with a small data set.

The implementation of CLIPSBase, a real-time database for [PI], is achieved by structuring the information based on the relational model and building tables in the form of facts in the CLIPS fact-list. This had the effect of providing a common pool of information on which different knowledge base modules can apply their reasoning. The functionality of CLIPSBase also provides a uniform set of access mechanisms using SQL syntax, giving greater organization and flexibility in manipulating data than the current distributed file system.

Of the real-time mechanisms implemented, sharing processes and results offers the greatest advantage because it eliminates much of the overhead involved when transfering data and processor control in the distributed system. Query optimization is effective in the sense that computation is simplified in the data retrieval process. File access minimization is based on the prefetching technique, which reduces or avoids swapping of information in and out of the computer's main memory.

When using the functions of CLIPSBase, some design considerations should always be kept in mind. The efficiency with which the DBMS operates depends to some extent on the size of the tables how the tables are set up in the fact-list. When it is possible, a database designer should break large tables down into smaller ones and avoid an excessive number of records. This does not mean that the database cannot contain much information. Rather, the information should be structured so that accesses can be performed more quickly.

The functions provided by CLIPSBase include, among others, *select*, *insert*, *update*, and *delete*. The *select* function allows portions of a table to be retrieved and asserted into the CLIPS fact-list as new tables. The *insert* function allows new tables and fields to be appended to existing tables. The *update* function allows modification of values in table records. The *delete* function allows deletion of records and fields from tables when it becomes necessary. Although the current implementation comes with a limited set of DBMS functions, it is a drastic improvement over the distributed file system, where at present there is no DBMS functionality, other than the ability to store and retrieve ASCII strings from text files.

For [PI], the implementation of CLIPSBase suffices for its real-time purposes. Compared to other real-time database systems, however, the access functions are relatively slow at data manipulation. One limitation of the database is that it is only appropriate for applications with small data sets. With larger data sets, the functions are found to perform rather poorly. For example, the *update* and *delete* functions require time overheads that are linear in relationship to the number of records in a table. This behavior would be inappropriate for a system like that used at American Express where a table can contain thousands of records. As a result, the implementation scheme described in this thesis would require enhancements to the access functions so that large tables can be handled more efficiently.

5.2 Directions for Future Research

This research addresses a number of ideas for providing DBMS functionality to knowledge shells that require a custom-built real-time database. These ideas, however, can be extended even further. This section describes five directions in which the work done here can be improved or enhanced by providing more efficient functions and building features on top of existing capabilities.

5.2.1 Tuple Evaluations

The current CLIPSBase implementation is not particularly strong at evaluating tuples, which comprise the relational algebra in the *where* clause. Some of the overhead time involved in manipulating data can thus be reduced if the a more efficient algorithm is used. Although the conditional clause does not impose length or complexity restrictions, this versatility was achieved at the expense of speed. For real-time applications, this versatility is probably not necessary. A future version of the *where* function may be reduced in terms of the number of rules required to evaluate it, which would improve execution time.

5.2.2 Higher Level Functionality

As mentioned before, the implementation of a DBMS in a knowledge shell allows for easy database extendibility. Functions that make the database smarter can be quickly built onto the data model. Some of these functions may include the ability to perform trend analysis on a series of runs, complicated reasoning processes that can lead to the conclusion that a certain set of data is “interesting”, or analysis on the amount of time consumed in each session to see which steps in the protocol tend to run over.

This type of functionality is the direction in which future architectures of information processing systems are going. Data and knowledge must interact to produce more information. With more information, a system becomes more intelligent and thus more powerful. CLIPSBase is a step in that direction for [PI] by organizing data more effectively and allowing a synergistic merge of the knowledge base and database.

For example with many object-oriented databases, performance has received a great deal of focus. In such systems, the performance is improved through extended functionality, that is by making the database more intelligent. With greater intelligence, they can make better decisions that influence their ability to perform well. As a result, greater efficiency is achieved not only through improvements in the procedures, but also through improvements in strategies used to execute these procedures.

5.2.3 Extended Storage of Evaluations

In the current implementation of CLIPSBase, only one set of *select* evaluations can be stored per table in the database. This imposes constraints on the database designer who must determine the way the database can be accessed that will reduce re-evaluation. With a multiple number of storage areas for *select* evaluations, such a constraint is lifted, allowing more flexibility in the sequence in which modules can retrieve and share data. Because of the limited data set, a large number of evaluations can be stored in the knowledge base simultaneously.

There are a number of ways this can be accomplished. Perhaps, evaluations can have indexes assigned to them. When a number of modules wish to access the same evaluation, they would simply refer to the index. Perhaps, some form of "smart" evaluation can be implemented like the "smart-loading" capability of the *load* function, which loads a table into memory only if it is not already there. Here, a module would issue a *select* statement and some rules would be applied to see if the evaluation already exists in the fact-list. If the evaluation was already performed, the statement would get ignored.

5.2.4 Alternatives

A design consideration during the development of CLIPSBase included the fact that some of the DBMS functionality can be implemented in the form of simple external commands, as extensions to the CLIPS shell. The performance improvement using this approach, however, is not known. The trade-offs must be considered in order to make these external commands work properly.

One advantage is that external commands are compiled functions, not interpreted functions like those in CLIPS, and can sometimes execute faster. Furthermore, special disk functions can be written. They would allow pointers to values in the text files and provide random access reading and writing. However, development of a database using C, the language used in making extensions in CLIPS, would require a substantially greater development time. Also, external commands cannot be ported directly from the Macintosh to other platforms, whereas the current CLIPSBase implementation is entirely within the CLIPS shell and can work on an IBM, a SUN using UNIX, a VAX using VMS, and any other platform for which CLIPS is available.

Moreover, the current implementation of CLIPSBase allows results of relational evaluations to be deposited directly into the fact-list. A DBMS using external functions can do this only with difficulty. Although they will run faster, extra overhead will be required in order to transfer evaluations performed by the external commands to the fact-list.

In order for external commands to work effectively, therefore, certain functionalities of CLIPSBase need to be judiciously selected. For example, search mechanisms currently used by CLIPSBase and fact manipulation rules can probably work faster using external commands. Also, tuple evaluation is another time consuming process that an external command can do more efficiently. On the other hand, extracting values from a list of facts is something that CLIPS rules can already do quite well. An external command would have a difficult time accessing the fact-list, extracting the values, and depositing the result back into the fact-list. These are alternatives that need further research.

5.2.5 Off-line HyperCard Interface

Finally, an off-line interface is needed for CLIPSBase. This would allow easier entering of data. Perhaps, a full-blown relational database can be implemented in HyperCard that would allow storage of views and more complicated relational commands like *join* and *project*. The purpose of this off-line software would not only be to provide more functionality for CLIPSBase, but also to give it the look and feel of a fully functional database.

Such an interface is intended to make off-line analysis of data much easier. Instead of typing commands into the CLIPS dialog window, a user could simply manipulate buttons, fields, and menu functions to generate different views. Afterwards, the information could be formatted, printed, and saved. The development of such an interface would be interesting to pursue.

All of these research directions would be exciting extensions to the investigations presented in this thesis. They make it possible for a generalization of CLIPSBase to many other applications.

REFERENCES

[1] *Apple HyperCard Script Language Guide: The HyperTalk Language*, Apple Computer Inc., 1988.

[2] Bhatnagar, R., Young, L. R., Groleau, N., Lam, C., Frainier, R., Compton, M., Manahan, M., "Principal Investigator in a Box: A Detailed Documentation," Man-Vehicle Laboratory, M.I.T., November 1990.

[3] Ceri, S., Gottlog, G., Wiederhold, G., "Efficient Database Access from Prolog", *IEEE Transactions on Software Engineering*, Vol. 15, No. 2, February 1989.

[4] Chu, W. W. and Sit, C., "Estimating Task Response Time with Contentions for Real-Time Distributed Systems," *IEEE Real-Time Systems Symposium*, 1989.

[5] *CLIPS Reference Manual*, Artificial Intelligence Section, Lyndon B. Johnson Space Center, April 1988.

[6] Corkill, D. D. and Dodhiawala, R., "Building Blackboard Applications," *AAAI Conference Tutorial*, Boston, Massachusetts, July 1990.

[7] Date, C. J., *Introduction to Database Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[8] DiPesa Jr., A. P., *Real-Time Extensions to a Relational Database*, Master of Science Thesis, Department of Electrical Engineering and Computer Science, M.I.T., May 1987.

[9] Feigenbaum, E., McCorduck, P., Nii, H. P., *The Rise of the Expert Company*, Times Books, New York, 1988.

[10] Friesen, O. and Golshani, F., "Databases in Large AI Systems," *AAAI Workshop*, Winter 1989.

[11] *GURU*, James Martin Productivity Series Report, Product Information, 1989.

[12] Hansson, H. and Jonsson, B., "A Framework for Reasoning about Time and Reliability," *IEEE Real-Time Systems Symposium*, 1989.

[13] Haymann-Haber, G., *PI in a Box: The Design of an "Expert" Protocol Manager*, MS/AI Practicum, Knowledge Systems Laboratory, Stanford University, June 1989.

[14] Hong, K. S. and Leung, Y., "On-Line Scheduling of Real-Time Tasks," *IEEE Real-Time Systems Symposium*, 1988.

[15] Lam, C., "HyperCLIPS," Documentation for Using HyperCLIPS, [PI] Team, M.I.T., 1990.

[16] Ullman, J. D., *Principles of Database and Knowledge-Base Systems Volume 1*, Computer Science Press, Rockville, Maryland, 1988.

[17] Walters, J. R. and Nielsen, N. R., *Crafting Knowledge-Based Systems: Expert Systems Made Easy (Realistic)*, John Wiley & Sons, New York, 1988.

[18] Wendorf, J. W., "Implementation and Evaluation of a Time-Driven Scheduling Processor," *IEEE Real-Time Systems Symposium*, 1988.

[19] Wiederhold, G., "Future Architectures for Information Processing Systems," Departments of Computer Science and Medicine Paper, Stanford University, January 8, 1990.

[20] Young, L. R., "[PI] 'Principal Investigator in a Box' Using Expert Systems for Manned Space Science," A Proposal to NASA Ames Research Center, July 1988.

[21] Young, L. R., Shelhamer, M., Modestino, S. A., "M.I.T./Canadian Vestibular Experiments on the Spacelab-1 Mission: 2. Visual Vestibular Interaction in Weightlessness," *Experimental Brain Research 64*, pp. 244-307, 1986.

[22] Young, L. R., Oman, C. M., Watt, D. G. D., Money, K. E., Lichtenberg, B. K., Kenyon, R. V., Arrott, A. P., "M.I.T./Canadian Vestibular Experiments on the Spacelab-1 Mission: 1. Sensory Adaptation to Weightlessness and Readaptation to One-g: an Overview," *Experimental Brain Research 64*, pp. 291-298, 1986.

Appendix A

CLIPSBase User's Guide

CLIPSBASE USER'S GUIDE

Introduction

CLIPSBase is a real-time relational database implemented in the CLIPS rule-based environment. CLIPSBase was written for the specific purpose of providing a relational database for expert systems developed in CLIPS. All of the basic features of a relational database are supported. This includes table creation, retrieval of a portion of a table, linked selection amongst a multiple number of tables, record and field insertion, record and field deletion, multi-parameter updates, and more. The instructions below on how to use CLIPSBase assume familiarity with CLIPS on the part of the developer.

Convention: In the following instructions on CLIPSBase, functions, rules, and information to be provided by the user are given in **Bold Helvetica**. Normal text is given in Plain Text Helvetica.

As a real-time database, CLIPSBase provides a single facility for viewing data. Using the **display-table** function, a CLIPS developer may display a table in the dialog window. CLIPSBase was designed to handle as many tables and records as the computer memory permits. It runs most quickly with tables containing 50 or fewer records. On a Macintosh SE/30, the **fast-select** function takes about 0.157 seconds to operate on a table 7 fields wide and 50 records long.

To use CLIPSBase, simply load parts 1 and 2 of the rules into memory. To implement a function, a fact is asserted into the fact-list according to the format explained below. All of the functions have salience values between 5000 and 6000. This means that CLIPSBase functions will execute before knowledge base rules with priority less than 5000. To make a knowledge base rule to fire before any CLIPSBase function executes, assign a priority of more than 6000 to the knowledge base rule. CLIPS allows salience values between -10000 and 10000. All CLIPSBase functions have the same priority. To operate correctly, different functions need be asserted into the fact-list in different knowledge base rules. For example, if the developer wished to assert the **create table** and **insert** functions, they need to be asserted within two different rules.

Normally, CLIPSBase functions are asserted into the fact-list by using the **assert** construct within a knowledge base rule. CLIPSBase functions can also operate on tables by asserting them through the dialog window and selecting **run (command-R)** from the Execution menu. To view results of operations performed in this manner, the developer may list the facts in the facts window, or display a table in the dialog window by asserting **(display-table <table-name>)** and selecting **run (command-R)** from the Execution menu.

CLIPSBase uses only five basic functions: **insert, select, update, delete,** and **where**. The other functions are merely utilities, variations of the five basic functions, or higher level procedures built out of the basic functions. Very versatile, complicated canned procedures can be written using the CLIPSBase functions listed below.

CONTENTS

create table

Usage: **(create table <table-name> fields <field-list> widths <width-list>)**

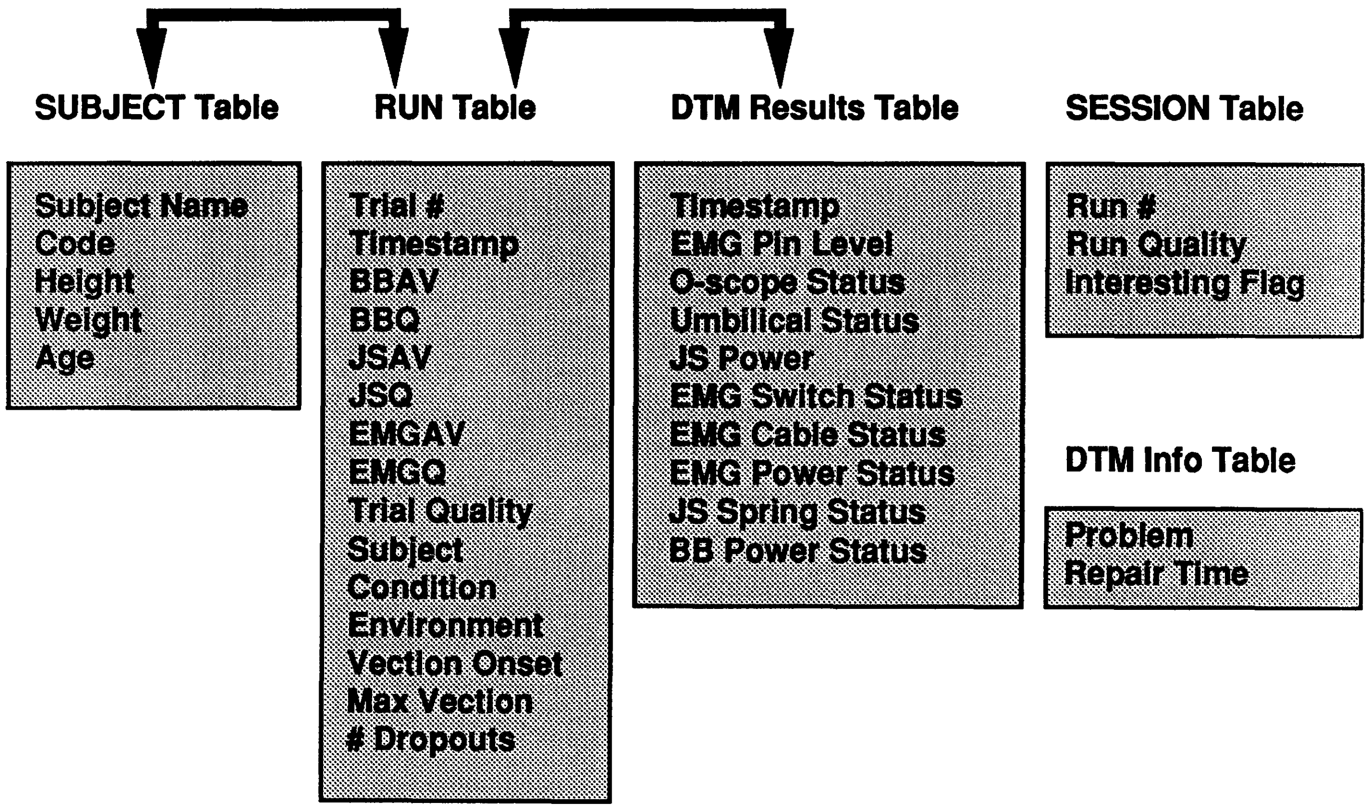
Before any table may be used, it must be created using **create table**. **<table-name>** must be a single word and should reflect its contents. A full path-name is normally used; otherwise, the file is created in the same folder in which the CLIPS application resides. When the table is created on disk, the suffix ".tbl" is appended to the end of the table name to indicate that the file is a CLIPSBase table. **<field-list>** is a list of field names separated by spaces. Multi-word field names need to use dashes (-)'s or underscores (_)'s to separate the words. **<width-list>** is a list of field widths, numbers separated by spaces, used solely for displaying the table in the dialog window when the **display-table** function is used. A field width is the number of character widths used for displaying a single field. Maximum width is 40. Field entries, including the field name, should not exceed the specified width or they will be displayed incorrectly.

Example:

```
(defrule set-up-database
     ?f1<-(CLIPSBase is already loaded)
     ?f2<-(set up pi-in-a-box tables for a single dome session)
=>
     (assert (create table SUBJECT fields Subject-Name Code
     Height Weight Age widths 20 6 8 8 5))
     (assert (create table RUN-n fields Timestamp Trial-# BBAV
     BBQ JSAV JSQ EMGAV EMGQ Trial-Quality Subject Condition
     Environment Vection-Onset Max-Vection #-Dropouts
     widths 12 8 6 6 6 6 6 6 15 9 20 20 17 15 13))
     (assert (create table DTM-Results fields Timestamp
          EMG-Pin-Level O-scope-Status Umbilical-Status
          JS-Power EMG-Switch-Status EMG-Cable-Status
          EMG-Power-Status JS-Spring-Status BB-Power-Status
          widths 20 20 20 20 20 20 20 20 20 20))
     (assert (create table SESSION fields Run-# Run-Quality
          Interesting-Flag widths 7 15 19))
     (assert (create table DTM-Info fields Problem Repair-Time
          widths 40 15))
     (retract ?f1 ?f2))
```

This sets up five tables (number of fields), SUBJECT (5), RUN-n (15), DTM-Results (10), SESSION (3), and DTM-Info (2), based upon the data model shown below. RUN tables can be created whenever needed, each table with a different name.

Data Model



SUBJECT Table

Subject Name
Code
Height
Weight
Age

RUN Table

Trial #
Timestamp
BBAV
BBQ
JSAV
JSQ
EMGAV
EMGQ
Trial Quality
Subject
Condition
Environment
Vection Onset
Max Vection
Dropouts

DTM Results Table

Timestamp
EMG Pin Level
O-scope Status
Umbilical Status
JS Power
EMG Switch Status
EMG Cable Status
EMG Power Status
JS Spring Status
BB Power Status

SESSION Table

Run #
Run Quality
Interesting Flag

DTM Info Table

Problem
Repair Time

insert

Usage: **(insert into <table-name> values <items-list>)**
or **(insert into <table-name> field <field-name> width <field-width>)**

To insert records into a table, use the first format. **<table-name>** specifies the table. **<items-list>** is a list of record items separated by spaces. Always insert the proper number of items within the number of character widths and in the order specified by **(<table-name> field-name <field-name-list>)** residing in the fact-list. Use an asterisk (*) in place of an unknown value, or else subsequent operations on the table will not work properly.

To insert new fields into a table, use the second format. **<table-name>** specifies the table. **<field-name>** is the name of the new field. The name cannot be the same as another field within a table. Multi-word fields must use dashes or underscores to separate the words. **<field-width>** is the number of character widths to be used for the **display-table** function. Since the values for the new field are initially unknown, asterisks are inserted into existing records.

Examples:

```
(defrule  insert-new-run-into-session-table
      ?f1<-(run ?n was completed)
      ?f2<-(parameters for run ?n are $?run-parameters)
=>
      (assert (insert into SESSION values ?n $?run-parameters))
      (retract ?f1 ?f2))

(defrule  insert-new-field-into-session-table
      ?f1<-(we need another field for session table)
      ?f2<-(because idf needs another parameter for its reasoning)
=>
      (assert (insert into SESSION field Run-Duration width 15))
      (retract ?f1 ?f2))
```

select

Usage: **(select <field-list> from <table-name>**
where <conditional-list>)
or **(select <field-list> from <table-name>)**

Use **select** to retrieve all or portion of a table. The second format differs from the first in that it has no conditional list attached, implying that all records in **<table-name>** are to be selected. The **<field-list>** is a list of field names to be included in the selection. All others are excluded. If an asterisk (*) is used in place of **<field-list>**, all fields are selected.

Only those records satisfying the **<conditional-list>** are retrieved. The **<conditional-list>** is a chain of conditionals which each follow the format:

<field-name> <comparator> <value>

where **<field-name>** is the name of the field under comparison and **<comparator>** is from the list:

eq, neq, <, <=, >, >=.

Conditionals are chained together using **and**'s and/or **or**'s. Parentheses are not allowed to force the evaluation sequence of **and**'s and **or**'s. The conditionals chained by **and**'s have higher precedence in the evaluation sequence than those chained by **or**'s.

Example: **Condition eq bungee or JSAV > 50 and JSAV <= 90**

Here, the conditional list is evaluated according to the order:

(Condition eq bungee) or ((JSAV > 50) and (JSAV <= 90)).

The **<conditional-list>** may be any length; however, a long and complicated list of conditionals takes more evaluation time than a short and simple one.

Once the selection is made, it is asserted into the fact-list according to the format:

```
(* <table-name> field-name <field-name-list>)
(* <table-name> field-width <field-width-list>)
(* <table-name> record <record-index> <record-items-list>)
(* other records)
```

The preceding asterisks (*) indicate that the facts are the result of a select function. The **<field-name-list>** is the list of field names under selection. **<field-width-list>** is the corresponding list of field widths. **<record-index>** reflects the order number of the record within its file, and is used as a primary key by a number of CLIPSBase functions.

In the following example where a diagnostic and troubleshooting module (dtm) is used to diagnose an equipment failure, a certain run was found to be bad and a dtm session is initiated to see if time is available for equipment repairs. The first rule retrieves information pertinent to the problem at hand and asserts the information into the fact-list. The second rule assumes that an unknown value, represented by an asterisk (*), means that there is nothing wrong. If a known value is given for the entry, then the device under scrutiny is assumed to be in need of repair. The third rule takes the results of the second rule and finds out if there is time available for repairing the ailing device(s). If time is available, the user is told to proceed with repairs.

Example:

```
(defrule  get-data-for-dtm-analysis
      ?f1<-(run ?n was bad)
      ?f2<-(dtm is initiated)
      (time when problem occurred is ?timestamp)
=>
      (assert (select Interesting-Flag from SESSION
                  where Run-# eq ?n))
      (assert (select * from DTM-Info))
      (assert (select * from DTM-Results
                  where Timestamp eq ?timestamp))
      (retract ?f1 ?f2))

(defrule  locate-sources-of-problem
      ?f1<-(time when problem occurred is ?timestamp)
      (* DTM-Results record ?index ?time-stamp ?emg-pin-level
      ?o-scope-status ?umbilical-status ?js-power
      ?emg-switch-status ?emg-cable-status ?emg-power-status
      ?js-spring-status ?bb-power-status)
      (* SESSION record ?index ?interesting-flag)
=>
      (if (eq ?interesting-flag off) then
            (if (or     (neq ?emg-pin-level *)
                        (neq ?emg-switch-status *)
                        (neq ?emg-power-status *)
                        (neq ?emg-cable-status *)) then
                  (assert (emg needs repair)))
            (if (neq ?o-scope-status *) then
                  (assert (o-scope needs repair)))
            (if (neq ?umbilical-status *) then
                  (assert (umbilical needs repair)))
            (if (or     (neq ?js-power *)
                        (neq ?js-spring-status *)) then
                  (assert (js needs repair)))
            (if (neq ?bb-power-status *) then
                  (assert (bb needs repair))))
```

```
        (retract ?f1))

(defrule check-repair-time-availability
        ?f1<-(?device needs repair)
        (* DTM-Info record ?index ?device ?repair-time)
        (time available for repair is ?time-available)
=>
        (if (<= ?repair-time ?time-available) then
                (printout t "Please repair the " ?device "." crlf))
        (retract ?f1))
```

Although in the preceding example there really was no need for using **select** to extract information from the relevant tables, it illustrates how a single value, a whole table, or a portion of a table can be selected and how historical experimental data can be recalled for reasoning.

update

Usage: **(update <table-name> set <set-list> where <conditional-list>)**

Use update to change one or more values in **<table-name>**. **<set-list>** is a list of set statements, each following the format:

<field-name> eq <value>

<set-list> may be any length. **<conditional-list>** follows the convention described under the **select** function.

Example:

```
(defrule update-session-table
     ?f1<-(run ?n was found to be interesting and of good quality)
=>
     (assert (update SESSION set Run-Quality eq good
          Interesting-Flag eq on where Run-# eq ?n))
     (retract ?f1))
```

delete

Usage: **(delete from <table-name> where <conditional-list>)**
or **(delete from <table-name> field <field-name>)**

To delete records from **<table-name>**, use the first format. Only those records satisfying the list of conditionals in **<conditional-list>** are deleted. **<conditional-list>** follows the convention described under the **select** function. Use the second format to delete a single field from **<table-name>**.

Examples:

```
(defrule delete-bad-run-from-session-table
     ?f1<-(delete run ?n in the session table)
=>
     (assert (delete from SESSION where Run-# eq ?n))
     (retract ?f1))

(defrule delete-field-from-subject-table
     ?f1<-(field ?field is not needed in subject table)
=>
     (assert (delete from SUBJECT field ?field))
     (retract ?f1))
```

where

Usage: **(in <table-name> where <conditional-list>)**

Although it is not normally used by the developer, the **where** function is available to retrieve those records that satisfy a list of conditionals. The table is specified by **<table-name>** and the list of conditions is specified by **<conditional-list>**, according to the convention explained under the **select** function. **<table-name>** must be a table already loaded into the fact-list. **where** returns those record indexes whose records that satisfy the conditional list. The result is asserted into the fact-list according to the format:

(<and/or-list> record-seq <record-indexes>)

The **<and/or-list>** is a list of **and**'s and **or**'s to be ignored by the developer. To extract the **<record-indexes>**, use a multi-field variable in the left-hand-side conditional. The salience value of the rule making the extraction must be at least 6000 because the record indexes are automatically retracted from the fact-list by CLIPSBase.

Example:

```
(defrule  get-records-in-run-n-with-bad-quality
     (dtm has been initiated)
=>
     (assert (in Run-n where Trial-Quality eq bad)))

(defrule  extract-record-indexes-with-bad-trial-quality
     (declare (salience 6000))
     (get trials with bad qualities)
     ?f1<-($?junk record-seq $?index-list)
=>
     (assert (the trials with bad quality have indexes $?index-list))
     (retract ?f1))
```

insert file

Usage: **(insert into <table-name> file <file-name>)**

insert file is a utility allowing insertion of records residing in an environment other than CLIPS, such as HyperCard. The program in HyperCard must save its records onto a file according to the format:

```
<field-list> <return>
<record-items-list> <return>
<other-records>
```

For example, a file called new-trial-parameters may contain:

```
Trial-# Timestamp BBAV BBQ JSAV JSQ EMGAV EMGQ Trial-Quality
4 t4*20*24 56 1 40 -1 46 0 0
5 t4*20*54 34 1 56 -1 67 0 1
```

<field-list> is a list of field names separated by spaces and terminated by **<return>**. **<record-items-list>** is a list of record items separated by spaces and terminated by **<return>**. Any number of records (two in the above example) using the same format may be included. Multi-word field names and record items must use underscores (_)'s or dashes (-)'s to separate words. **insert file** inserts only those fields that correspond exactly with those in **<table-name>**. All others are ignored and missing information is padded with asterisks. Insertion of records is done without checking if the records already reside in **<table-name>**.

Example:

```
(defrule insert-info-from-hypercard-into-run-n
      (run-n has completed)
      ?f1<-(get info from hypercard)
=>
      (assert (insert into Run-n file new-trial-parameters))
      (retract ?f1))
```

fast-select

Usage: **(fast-select <field-name> from <table-name> where <field-name> eq <value>)**

An optimized **select** function for the specific purpose of extracting a single value from a database is provided. **fast-select** retrieves a single value from **<table-name>** and asserts a fact according to the format:

(fast-select: <value>)

<value> may be the empty set. Unlike the normal **select** function, only one conditional using the **eq** comparator is allowed.

Example:

The example here queries the database for information. In the event the answer is unknown, represented by an asterisk (*), the query is posed to the astronaut user and the response is saved into the database.

```
(defrule  query-the-database-for-information
      (was the trial quality good for trial ?n)
=>
      (assert (fast-select Trial-Quality from Run-n
                  where Trial-# eq ?n)))

(defrule  auto-query-the-astronaut
      (auto query on)
      (was the trial quality good for trial ?n)
      ?f1<-(fast select: ?value)
=>
      (if (eq ?value *) then
            (printout t "What was the trial quality for trial " ?n "?" crlf)
            (bind ?answer (read))
            (assert (update Run-n set Trial-Quality eq ?answer
                        where Trial-# eq ?n)))
      (retract ?f1))
```

linked-select

Usage: **(linked-select <field-list> from <table-name> where <fast-select-conditional-list>)**

A special **select** function called **linked-selection** is provided to allow linked selection using one of more values from different tables. **<field-list>** is a list of field names to be included in the resulting retrieval operation. **<table-name>** specifies the table from which the fields are taken. **<fast-select-conditional-list>** is a list of conditionals, each following the format:

```
<field-name>  <comparator>
                              fast-select  <field-name>
                              from  <table-name>
                              where  <field-name>  eq  <value>
```

<fast-select-conditional-list> may be a combination of conditions joined by **and**'s and **or**'s as described above for **<conditional-list>** under the **select** function.

The first **<field-name>** is a field name from the first **<table-name>**. **<comparator>** is from the list:

eq, neq, <, <=, >, >=.

The **fast-select** format follows the standard convention described under the **fast-select** function. The second **<table-name>** may be any table, including the first **<table-name>**. The **<field-name>**'s following **fast-select** refer to the second **<table-name>** in the above format. Like the normal **select** function, the selection is asserted into the fact-list according to the format described under the **select** function.

Example:

```
(defrule  retrieve-dam/dqm-parameters-for-subject
     ?f1<-(retrieve  dam/dqm  results  for  subject  ?subject-code)
=>
     (assert (linked-select BBAV BBQ JSAV JSQ EMGAV EMGQ
                    from Run-n
                    where  Subject  eq
                                   fast-select  Subject-Name
                                   from  SUBJECT
                                   where  Code  eq  ?subject-code))
     (retract  ?f1))
```

display-table

Usage: **(display-table <table-name>)**

display-table displays the table specified by **<table-name>** in the Dialog window of CLIPS. As a real-time database, CLIPSBase provides **display-table** only for development purposes.

Example:

```
(defrule display-the-session-table
      ?f1<-(display session table)
=>
      (assert (display-table SESSION))
      (retract ?f1))
```

load-table

Usage: **(load-table <table-name>)**

Manual loading of a table is possible using **load-table**. **load-table** loads **<table-name>** into the fact-list only if the table is not already there. Certain CLIPSBase functions automatically execute the **load-table** command and do not require the developer to assert **load-table** before accessing the database. These functions are:

- **insert**
- **select**
- **update**
- **delete**
- **insert file**
- **fast-select**
- **linked-select**
- **display-table**

Example:

```
(defrule  load-all-tables-into-memory
        ?f1<-(load  all  tables  into  memory)
=>
        (assert  (load-table  SUBJECT))
        (assert  (load-table  RUN-n))
        (assert  (load-table  DTM-Results))
        (assert  (load-table  SESSION))
        (assert  (load-table  DTM-Info))
        (retract  ?f1))
```

do-load-table

Usage: **(do-load-table <table-name>)**

Although not normally used by the developer, **do-load-table** is provided so that a table may be loaded into memory without checking to see if it is already there. This is particularly useful if a **user-defined insert** function is implemented to insert records into a CLIPSBase table file.

Example:

```
(defrule my-insert-function
      ?f1<-(insert ?my-record into ?table-name-file)
=>
      (assert (user-defined insert into ?table-name-file
                    record ?my-record))
      (assert (do-load-table ?table-name-file))
      (retract ?f1))
```

save-table

Usage: **(save-table <table-name>)**

Although not normally used by the developer, save-table is provided to allow manual saving of tables into their CLIPSBase table files. This is particularly useful for user-defined functions that modify the database. All CLIPSBase functions that modify the database automatically save their results onto table files.

Example:

```
(defrule save-my-custom-modifications-to-session-table
     ?f1<-(user-defined functions have completed their
          changes to session)
=>
     (assert (save-table SESSION))
     (retract ?f1))
```

retract-table

Usage: **(retract-table <table-name>)**

Sometimes it may be necessary to remove a table resulting from a select evaluation from memory. **retract-table** is provided to do just that.

Example:

```
(defrule retract-subject-table
      ?f1<-(subject table is no longer needed)
=>
      (assert (retract-table SUBJECT))
      (retract ?f1))
```

remove-table

Usage: **(remove-table <table-name>)**

Sometimes it may be necessary to remove a table in the fact-list. **remove-table** is provided to do just that.

Example:

```
(defrule remove-subject-table
     ?f1<-(subject table is no longer needed)
=>
     (assert (retract-table SUBJECT))
     (retract ?f1))
```

Appendix B

CLIPSBase Source Code

```
;*************************************************************************
;           CLIPSBASE 1.0:  A Real-time Relational Database
;                            by Sen-Hao Lai
;
;           Simply load parts 1 & 2 of the rules into memory.
;*************************************************************************

;*************************************************************************
;******************************** PART 1 *********************************
;*************************************************************************

;*************************************************************************
;CREATE TABLE:  Creates a new table specified by <table-name>.
;Usage:  create table <table-name> fields <field-list> widths <width-list>
;Multi-word field names must use dashes or underscores.
;*************************************************************************
(defrule create-table
     (declare (salience 5000))
     ?f1<-(create table ?table-name fields $?field-names widths
           $?field-widths)
=>
     (open (str-cat ?table-name ".tbl") file-tag "w")
     (printout file-tag $?field-names crlf)
     (assert (?table-name field-name $?field-names))
     (printout file-tag $?field-widths crlf)
     (assert (?table-name field-width $?field-widths))
     (close file-tag)
     (retract ?f1))

;*************************************************************************
;LINKED-SELECT:  Allows linked selection using one or more values
;                taken from different tables.
;Usage:  linked-select <field-list> from <table-name> where <conditional-list>
;<conditional-list> follows the format:  <field-name> <comparator> fast-select
;                                        <field-name> from <table-name> where
;                                        <field-name> eq <value>
;<conditional-list> may be any length.
;<comparator> is from the list:  eq, neq, <, <=, >, >=.
;*************************************************************************
(defrule linked-select
     (declare (salience 5100))
     ?f1<-(linked-select $?blurb1 fast-select ?field1 from ?table-name
           where ?field2 eq ?value $?blurb2)
=>
     (retract ?f1)
     (assert (fast-select ?field1 from ?table-name where ?field2 eq
              ?value))
     (assert (linked-select $?blurb1 fast-select-function $?blurb2)))

(defrule get-fast-select-function
     (declare (salience 5100))
     ?f1<-(linked-select $?fields from ?table-name $?blurb1
           fast-select-function $?blurb2)
     ?f2<-(fast-select: $?value)
=>
     (if (> (length $?value) 0) then
         (assert (linked-select $?fields from ?table-name $?blurb1
                  $?value $?blurb2)) else
```

```
          (assert (select * from ?table-name)))
     (retract ?f1 ?f2))

(defrule get-select-function
     (declare (salience 5000))
     ?f1<-(linked-select $?blurb)
=>
     (retract ?f1)
     (assert (select $?blurb)))

;*********************************************************************
;FAST-SELECT:  Extracts a single value from <table-name> and asserts the
;              fact (fast-select: <value>).  <value> may be the empty set.
;Usage:  fast-select <field-name> from <table-name> where <field-name>
;        eq <value>
;*********************************************************************
(defrule fast-select
     (declare (salience 5400))
     (fast-select ?field1 from ?table-name where ?field2 eq ?value)
=>
     (assert (fast-select:))
     (assert (load-table ?table-name)))

(defrule fast-select-setup
     (declare (salience 5400))
     (fast-select ?field1 from ?table-name where ?field2 eq ?value)
     ?f1<-(fast-select: ?old-value)
=>
     (retract ?f1)
     (assert (fast-select:)))

(defrule get-field-number
     (declare (salience 5350))
     (fast-select ?field1 from ?table-name where ?field2 eq ?value)
     (?table-name field-name $?field-names)
=>
     (bind ?field-number2 (member ?field2 $?field-names))
     (bind ?field-number1 (member ?field1 $?field-names))
     (if (and (> ?field-number2 0)
              (> ?field-number1 0)) then
         (assert (field-number1 ?field-number1 field-number2
                   ?field-number2))))

(defrule compare-record-value
     (declare (salience 5350))
     ?f1<-(fast-select ?field1 from ?table-name where ?field2 eq ?value)
     ?f2<-(field-number1 ?field-number1 field-number2 ?field-number2)
     ?f3<-(fast-select:)
     (?table-name record ?index $?record-items)
=>
     (bind ?record-item1 (nth ?field-number1 $?record-items))
     (bind ?record-item2 (nth ?field-number2 $?record-items))
     (if (eq ?value ?record-item2) then
         (assert (fast-select: ?record-item1))
         (retract ?f1 ?f2 ?f3)))

(defrule clean-up-fast-select
     (declare (salience 5300))
     ?f1<-(fast-select ?field1 from ?table-name where ?field2 eq ?value)
=>
     (retract ?f1))

(defrule clean-up-field-number1/2
     (declare (salience 5300))
     ?f1<-(field-number1 ?field-number1 field-number2 ?field-number2)
=>
     (retract ?f1))

;************************************************************************
;INSERT:  Allows insertion of new records and fields into the table
;         specified by <table-name>.
;Usage:  insert into <table-name> values <items-list>
;   or   insert into <table-name> field <field-name> width <field-width>
;Multi-word items must use dashes or underscores.
;************************************************************************
(defrule insert-record
     (declare (salience 5000))
     ?f1<-(insert into ?table-name values $?items)
=>
     (open (str-cat ?table-name ".tbl") file-tag "a")
     (printout file-tag $?items crlf)
     (close file-tag)
     (assert (do-load-table ?table-name))
     (retract ?f1))

(defrule insert-field
     (declare (salience 5000))
     (insert into ?table-name field ?field-name width ?field-width)
=>
     (assert (load-table ?table-name))
     (assert (insert-field-into-field-items ?table-name))
     (assert (insert-field-into-records ?table-name))
     (assert (save-table ?table-name)))

(defrule clean-up-insert-field
     (declare (salience 5125))
     ?f1<-(insert into ?table-name field ?field-name width ?field-width)
     ?f2<-(insert-field-into-records ?table-name)
=>
     (retract ?f1 ?f2))

(defrule insert-field-into-field-items
     (declare (salience 5150))
     (insert into ?table-name field ?field-name width ?field-width)
     ?f1<-(insert-field-into-field-items ?table-name)
     ?f2<-(?table-name field-name $?field-items)
     ?f3<-(?table-name field-width $?width-items)
=>
     (retract ?f1 ?f2 ?f3)
     (assert (?table-name field-name $?field-items ?field-name))
     (assert (?table-name field-width $?width-items ?field-width)))

(defrule insert-field-into-records
     (declare (salience 5150))
     (insert-field-into-records ?table-name)
     ?f1<-(?table-name record $?record-items)
```

```
=>
     (retract ?f1)
     (assert (?table-name converted-record $?record-items *)))

(defrule unconvert-records
     (declare (salience 5110))
     ?f1<-(?table-name converted-record $?record-items)
=>
     (assert (?table-name record $?record-items))
     (retract ?f1))

;*********************************************************************
;RETRACT-TABLE:  Retracts a table specified by <table-name> currently
;                in the fact-list that was the result of a selection.
;Usage:  retract-table <table-name>
;*********************************************************************
(defrule retract-*-table
     (declare (salience 5600))
     ?f1<-(retract-table ?table-name)
     ?f2<-(* ?table-name field-name $?field-items)
     ?f3<-(* ?table-name field-width $?width-items)
=>
     (assert (retract-*-records ?table-name))
     (retract ?f1 ?f2 ?f3))

(defrule clean-up-*-retract-records
     (declare (salience 5500))
     ?f1<-(retract-*-records ?table-name)
=>
     (retract ?f1))

(defrule clean-up-*-retract
     (declare (salience 5500))
     ?f1<-(retract-table ?table-name)
=>
     (retract ?f1))

(defrule retract-*-records
     (declare (salience 5600))
     (retract-*-records ?table-name)
     ?f1<-(* ?table-name record $?record-items)
=>
     (retract ?f1))

;*********************************************************************
;REMOVE-TABLE:  Removes a table specified by <table-name> currently
;               in the fact-list.
;Usage:  remove-table <table-name>
;*********************************************************************
(defrule remove-table
     (declare (salience 5800))
     ?f1<-(remove-table ?table-name)
     ?f2<-(?table-name field-name $?field-items)
     ?f3<-(?table-name field-width $?width-items)
=>
     (assert (remove-records ?table-name))
     (retract ?f1 ?f2 ?f3))
```

```
(defrule clean-up-remove-records
     (declare (salience 5700))
     ?f1<-(remove-records ?table-name)
=>
     (retract ?f1))

(defrule clean-up-remove
     (declare (salience 5700))
     ?f1<-(remove-table ?table-name)
=>
     (retract ?f1))

(defrule remove-records
     (declare (salience 5800))
     (remove-records ?table-name)
     ?f1<-(?table-name record $?record-items)
=>
     (retract ?f1))

;***********************************************************************
;LOAD-TABLE:  Loads a table specified by <table-name> from a file into
;             memory if the table is not already in the fact-list.
;Usage:  load-table <table-name>
;***********************************************************************
(defrule check-load-table
     (declare (salience 5525))
     ?f1<-(load-table ?table-name)
     (?table-name field-name $?fields)
=>
     (retract ?f1))

(defrule load-table
     (declare (salience 5500))
     ?f1<-(load-table ?table-name)
=>
     (open (str-cat ?table-name ".tbl") file-tag "r")
     (str-assert (str-cat ?table-name " field-name "
                 (readline file-tag)))
     (str-assert (str-cat ?table-name " field-width "
                 (readline file-tag)))
     (bind ?file-input (readline file-tag))
     (bind ?index 0)
     (while (neq ?file-input EOF)
            (bind ?index (+ ?index 1))
                 (str-assert (str-cat ?table-name " record " ?index
                             " " ?file-input))
            (bind ?file-input (readline file-tag)))
     (close file-tag)
     (retract ?f1))

;***********************************************************************
;DO-LOAD-TABLE:  Loads a table into the fact-list without checking if it
;                is already there.
;Usage:  do-load-table <table-name>
;***********************************************************************
(defrule do-load-table
     (declare (salience 5500))
     ?f1<-(do-load-table ?table-name)
```

```
=>
     (open (str-cat ?table-name ".tbl") file-tag "r")
     (str-assert (str-cat ?table-name " field-name "
                  (readline file-tag)))
     (str-assert (str-cat ?table-name " field-width "
                  (readline file-tag)))
     (bind ?file-input (readline file-tag))
     (bind ?index 0)
     (while (neq ?file-input EOF)
            (bind ?index (+ ?index 1))
                 (str-assert (str-cat ?table-name " record " ?index
                  " " ?file-input))
            (bind ?file-input (readline file-tag)))
     (close file-tag)
     (retract ?f1))

;**********************************************************************
;SAVE-TABLE:  Saves the table specified by <table-name> in the fact-list
;             into its file.
;Usage:  save-table <table-name>
;**********************************************************************
(defrule save-table
     (declare (salience 5000))
     ?f1<-(save-table ?table-name)
     (?table-name field-name $?fields)
     (?table-name field-width $?widths)
=>
     (retract ?f1)
     (open (str-cat ?table-name ".tbl") file-tag "w")
     (printout file-tag $?fields crlf $?widths crlf)
     (assert (save-records ?table-name)))

(defrule save-records
     (declare (salience 5010))
     (save-records ?table-name)
     (?table-name record ?index $?record-items)
=>
     (printout file-tag $?record-items crlf))

(defrule clean-up-save
     (declare (salience 5000))
     ?f1<-(save-records ?table-name)
=>
     (retract ?f1)
     (close))

;**********************************************************************
;DELETE:  Deletes a record or field from the table specified by <table-name>.
;Usage:  delete from <table-name> where <conditional-list>
;   or   delete from <table-name> field <field-name>
;<conditional-list> follows the format:  <field-name> <comparator> <value>
;<comparator> is from the list:  eq, neq, <, <=, >, >=.
;**********************************************************************
(defrule delete
     (declare (salience 5100))
     (delete from ?table-name where $?conditional)
=>
     (assert (load-table ?table-name))
     (assert (in ?table-name where $?conditional)))

(defrule delete-record
     (declare (salience 5050))
     (delete from ?table-name where $?conditional)
     ($?operators record-seq $?sequence)
     ?f1<-(?table-name record ?index $?record-items)
=>
     (if (> (member ?index $?sequence) 0) then
         (retract ?f1)))

(defrule clean-up-delete-record
     (declare (salience 5020))
     ?f1<-(delete from ?table-name where $?conditional)
     ?f2<-($?operators record-seq $?sequence)
=>
     (retract ?f1 ?f2)
     (assert (save-table ?table-name))
     (assert (reload-table ?table-name)))

(defrule reload-table
     ?f1<-(reload-table ?table-name)
=>
     (retract ?f1)
     (assert (remove-table ?table-name))
     (assert (load-table ?table-name)))

(defrule delete-field
     (declare (salience 5200))
     (delete from ?table-name field ?field)
=>
     (assert (load-table ?table-name)))

(defrule delete-field-from-field-items
     (declare (salience 5140))
     ?f1<-(delete from ?table-name field ?field)
     ?f2<-(?table-name field-name $?names)
     ?f3<-(?table-name field-width $?widths)
=>
     (bind ?index (member ?field $?names))
     (if (> ?index 0) then
         (bind ?name-string (str-implode (mv-delete ?index
                                           $?names)))
         (str-assert (str-cat ?table-name " field-name "
                      ?name-string))
         (bind ?width-string (str-implode (mv-delete ?index
                                            $?widths)))
         (str-assert (str-cat ?table-name " field-width "
                      ?width-string))
         (retract ?f2 ?f3)
         (assert (save-table ?table-name)))
     (retract ?f1))

(defrule delete-field-from-record
     (declare (salience 5150))
     (delete from ?table-name field ?field)
     (?table-name field-name $?names)
     ?f1<-(?table-name record ?record-index $?items)
```

```
=>
     (bind ?index (member ?field $?names))
     (if (> ?index 0) then
         (bind ?record-string (str-implode (mv-delete ?index $?items)))
         (str-assert (str-cat ?table-name " converted-record "
                      ?record-index " " ?record-string))
         (retract ?f1)))

;*********************************************************************
;UPDATE:  Updates the table specified by <table-name> with new values.
;Usage:  update <table-name> set <set-list> where <conditional-list>
;<set-list> may be any length and follows the format:  <field-name> eq <value>
;<conditional-list> follows the format:  <field-name> <comparator> <value>
;*********************************************************************
(defrule update
     (declare (salience 5000))
     (update ?table-name set $?fields/values where $?conditional)
=>
     (assert (load-table ?table-name))
     (assert (in ?table-name where $?conditional))
     (assert (extract-fields/values $?fields/values))
     (assert (new-value-seq))
     (assert (field-number-seq)))

(defrule get-new-values
     (declare (salience 5250))
     ?f1<-(extract-fields/values $?set-statements ?field eq ?new-value)
     ?f2<-(new-value-seq $?new-values)
     ?f3<-(field-number-seq $?sequence)
     (?table-name field-name $?field-names)
=>
     (retract ?f1)
     (assert (extract-fields/values $?set-statements))
     (bind ?field-index (member ?field $?field-names))
     (if (> ?field-index 0) then
         (assert (new-value-seq ?new-value $?new-values))
         (assert (field-number-seq ?field-index $?sequence))
         (retract ?f2 ?f3)))

(defrule update-records
     (declare (salience 5200))
     (update ?table-name set $?fields/values where $?conditional)
     (field-number-seq $?field-seq)
     (new-value-seq $?new-values)
     ($?junk record-seq $?record-seq)
     ?f1<-(?table-name record ?index $?record-items)
=>
     (bind ?sequence-length (length $?field-seq))
     (if (and (> (member ?index $?record-seq) 0)
              (> ?sequence-length 0)) then
         (retract ?f1)
         (bind ?record-length (length $?record-items))
         (bind ?loop-index 1)
         (bind ?string (str-cat ?table-name " converted-record "
                        ?index))
         (while (<= ?loop-index ?record-length)
              (bind ?index-location (member ?loop-index
                                    $?field-seq))
```

```
                (if (> ?index-location 0) then
                    (bind ?string (str-cat ?string " " (nth
                     ?index-location $?new-values)))
                     else
                    (bind ?string (str-cat ?string " " (nth
                     ?loop-index $?record-items))))
                (bind ?loop-index (+ ?loop-index 1)))
         (str-assert ?string)))

(defrule clean-up-update
     (declare (salience 5150))
     ?f1<-(update ?table-name set $?fields/values where $?conditional)
     ?f2<-(extract-fields/values)
     ?f3<-(new-value-seq $?new-values)
     ?f4<-(field-number-seq $?numbers)
     ?f5<-($?junk record-seq $?record-seq)
=>
     (retract ?f1 ?f2 ?f3 ?f4 ?f5)
     (assert (save-table ?table-name)))

;************************************************************************
;SELECT:  Selects from the table specified by <table-name> and asserts
;         the selection into the fact-list.
;The asserted selection follows the format:
;         (* <table-name> field-name <field-name-list>)
;         (* <table-name> field-width <field-width-list>)
;         (* <table-name> record <record-index> <record-items-list>)
;         (* other records)
;<record-index> reflects the number of the record within the file.
;Usage:  select <field-list> from <table-name> where <conditional-list>
;   or   select <field-list> from <table-name>
;<conditional-list> follows the format:  <field-name> <comparator> <value>
;The symbol, *, in place of <field-list> selects all fields.
;The second format selects all records.
;************************************************************************
(defrule select
     (declare (salience 5000))
     ?f1<-(select $?selected-fields from ?table-name
           where $?conditional)
=>
     (assert (retract-table ?table-name))
     (assert (load-table ?table-name))
     (assert (in ?table-name where $?conditional))
     (assert (crop-table-records ?table-name))
     (assert (get-field-number-seq ?table-name))
     (assert (field-seq $?selected-fields))
     (retract ?f1))

(defrule select-*
     (declare (salience 5200))
     (select $?fields from ?table-name)
=>
     (assert (load-table ?table-name))
     (assert (retract-table ?table-name))
     (if (neq $?fields (str-explode "*")) then
         (assert (get-field-number-seq ?table-name))
         (assert (field-seq $?fields))
         (assert (sort-record-items ?table-name))))
```

```
(defrule select-*-record
     (declare (salience 5100))
     (select $?fields from ?table-name)
     (?table-name record $?record-items)
=>
     (assert (* ?table-name record $?record-items)))

(defrule select-*-names/widths
     (declare (salience 5050))
     ?f1<-(select $?fields from ?table-name)
     (?table-name field-name $?names)
     (?table-name field-width $?widths)
=>
     (if (eq $?fields (str-explode "*")) then
         (assert (* ?table-name field-name $?names))
         (assert (* ?table-name field-width $?widths)))
     (retract ?f1))

(defrule clean-up-record-seq
     (declare (salience 5000))
     ?f1<-($?operator record-seq $?record-numbers)
=>
     (retract ?f1))

(defrule clean-up-crop-table-records
     (declare (salience 5000))
     ?f1<-(crop-table-records ?table-name)
=>
     (retract ?f1))

(defrule clean-up-sort-record-items
     (declare (salience 5025))
     ?f1<-(sort-record-items ?table-name)
=>
     (retract ?f1))

(defrule clean-up-field-number-seq
     (declare (salience 5000))
     ?f1<-(field-number-seq $?numbers)
=>
     (retract ?f1))

(defrule crop-table-records
     (declare (salience 5100))
     ($?operator record-seq $?record-numbers)
     (crop-table-records ?table-name)
     (?table-name record ?index $?items)
=>
     (if (> (member ?index $?record-numbers) 0) then
         (assert (* ?table-name record ?index $?items))))

(defrule get-field-number-seq
     (declare (salience 5050))
     ?f1<-(get-field-number-seq ?table-name)
     ?f2<-(field-seq $?selected-fields)
     (?table-name field-name $?available-fields)
     (?table-name field-width $?widths)
=>
     (if (neq $?selected-fields (str-explode "*")) then
         (bind ?number-fields (length $?selected-fields))
         (bind ?field-index 1)
         (bind ?string "field-number-seq")
         (while (<= ?field-index ?number-fields)
                (bind ?next-word (nth ?field-index $?selected-fields))
                (bind ?field-location (member ?next-word
                                             $?available-fields))
                (if (> ?field-location 0) then
                    (bind ?string (str-cat ?string " "
                                         ?field-location)))
                (bind ?field-index (+ ?field-index 1)))
         (str-assert ?string)
         (assert (sort-width-seq ?table-name))
         (assert (sort-name-seq ?table-name))
         (assert (sort-record-items ?table-name)) else
         (assert (* ?table-name field-name $?available-fields))
         (assert (* ?table-name field-width $?widths)))
     (retract ?f1 ?f2))

(defrule sort-width-seq
     (declare (salience 5050))
     ?f1<-(sort-width-seq ?table-name)
     (?table-name field-width $?widths)
     (field-number-seq $?numbers)
=>
     (bind ?number-count (length $?numbers))
     (if (> ?number-count 0) then
         (bind ?index 1)
         (bind ?string (str-cat "* " ?table-name " field-width"))
         (while (<= ?index ?number-count)
                (bind ?next-width (nth (nth ?index $?numbers)
                                     $?widths))
                (bind ?string (str-cat ?string " " ?next-width))
                (bind ?index (+ ?index 1)))
         (str-assert ?string) else
         (assert (* ?table-name field-width)))
     (retract ?f1))

(defrule sort-name-seq
     (declare (salience 5050))
     ?f1<-(sort-name-seq ?table-name)
     (?table-name field-name $?names)
     (field-number-seq $?numbers)
=>
     (bind ?number-count (length $?numbers))
     (if (> ?number-count 0) then
         (bind ?index 1)
         (bind ?string (str-cat "* " ?table-name " field-name"))
         (while (<= ?index ?number-count)
                (bind ?next-name (nth (nth ?index $?numbers) $?names))
                (bind ?string (str-cat ?string " " ?next-name))
                (bind ?index (+ ?index 1)))
         (str-assert ?string) else
         (assert (* ?table-name field-name)))
     (retract ?f1))
```

```
(defrule sort-record-items
     (declare (salience 5050))
     (sort-record-items ?table-name)
     ?f1<-(* ?table-name record ?record-index $?record-items)
     (field-number-seq $?numbers)
=>
     (bind ?number-count (length $?numbers))
     (retract ?f1)
     (if (> ?number-count 0) then
         (bind ?index 1)
         (bind ?string (str-cat "* " ?table-name " converted-record "
                        ?record-index))
         (while (<= ?index ?number-count)
                (bind ?next-item (nth (nth ?index $?numbers)
                                 $?record-items))
                (bind ?string (str-cat ?string " " ?next-item))
                (bind ?index (+ ?index 1)))
         (str-assert ?string) else
         (assert (* ?table-name converted-record ?record-index))))

(defrule convert-sort-record
     (declare (salience 5000))
     ?f1<-(* ?table-name converted-record $?items)
=>
     (retract ?f1)
     (assert (* ?table-name record $?items)))

;*********************************************************************
;******************************* PART 2 ******************************
;*********************************************************************

;*********************************************************************
;DISPLAY-TABLE:  Displays the table specified by <table-name> in the
;                dialog window.
;Usage:  display-table <table-name>
;*********************************************************************
(defrule display-table
     (declare (salience 5000))
     ?f1<-(display-table ?table-name)
=>
     (assert (display-field-names ?table-name))
     (assert (display-records ?table-name))
     (assert (retract-display-records))
     (assert (load-table ?table-name))
     (retract ?f1))

(defrule display-field-names
     (declare (salience 5100))
     ?f1<-(display-field-names ?table-name)
     (?table-name field-name $?field-names)
     (?table-name field-width $?field-widths)
=>
     (bind ?number-fields (length $?field-names))
     (bind ?field-index 1)
     (while (<= ?field-index ?number-fields)
            (bind ?nth-field-name (str-cat (nth ?field-index
                                           $?field-names)))
            (bind ?nth-field-width (nth ?field-index $?field-widths))
```

```
(if (= ?nth-field-width 1) then
    (format t "%1s" ?nth-field-name))
(if (= ?nth-field-width 2) then
    (format t "%2s" ?nth-field-name))
(if (= ?nth-field-width 3) then
    (format t "%3s" ?nth-field-name))
(if (= ?nth-field-width 4) then
    (format t "%4s" ?nth-field-name))
(if (= ?nth-field-width 5) then
    (format t "%5s" ?nth-field-name))
(if (= ?nth-field-width 6) then
    (format t "%6s" ?nth-field-name))
(if (= ?nth-field-width 7) then
    (format t "%7s" ?nth-field-name))
(if (= ?nth-field-width 8) then
    (format t "%8s" ?nth-field-name))
(if (= ?nth-field-width 9) then
    (format t "%9s" ?nth-field-name))
(if (= ?nth-field-width 10) then
    (format t "%10s" ?nth-field-name))
(if (= ?nth-field-width 11) then
    (format t "%11s" ?nth-field-name))
(if (= ?nth-field-width 12) then
    (format t "%12s" ?nth-field-name))
(if (= ?nth-field-width 13) then
    (format t "%13s" ?nth-field-name))
(if (= ?nth-field-width 14) then
    (format t "%14s" ?nth-field-name))
(if (= ?nth-field-width 15) then
    (format t "%15s" ?nth-field-name))
(if (= ?nth-field-width 16) then
    (format t "%16s" ?nth-field-name))
(if (= ?nth-field-width 17) then
    (format t "%17s" ?nth-field-name))
(if (= ?nth-field-width 18) then
    (format t "%18s" ?nth-field-name))
(if (= ?nth-field-width 19) then
    (format t "%19s" ?nth-field-name))
(if (= ?nth-field-width 20) then
    (format t "%20s" ?nth-field-name))
(if (= ?nth-field-width 21) then
    (format t "%21s" ?nth-field-name))
(if (= ?nth-field-width 22) then
    (format t "%22s" ?nth-field-name))
(if (= ?nth-field-width 23) then
    (format t "%23s" ?nth-field-name))
(if (= ?nth-field-width 24) then
    (format t "%24s" ?nth-field-name))
(if (= ?nth-field-width 25) then
    (format t "%25s" ?nth-field-name))
(if (= ?nth-field-width 26) then
    (format t "%26s" ?nth-field-name))
(if (= ?nth-field-width 27) then
    (format t "%27s" ?nth-field-name))
(if (= ?nth-field-width 28) then
    (format t "%28s" ?nth-field-name))
(if (= ?nth-field-width 29) then
    (format t "%29s" ?nth-field-name))
```

```
              (if (= ?nth-field-width 30) then
                  (format t "%30s" ?nth-field-name))
              (if (= ?nth-field-width 31) then
                  (format t "%31s" ?nth-field-name))
              (if (= ?nth-field-width 32) then
                  (format t "%32s" ?nth-field-name))
              (if (= ?nth-field-width 33) then
                  (format t "%33s" ?nth-field-name))
              (if (= ?nth-field-width 34) then
                  (format t "%34s" ?nth-field-name))
              (if (= ?nth-field-width 35) then
                  (format t "%35s" ?nth-field-name))
              (if (= ?nth-field-width 36) then
                  (format t "%36s" ?nth-field-name))
              (if (= ?nth-field-width 37) then
                  (format t "%37s" ?nth-field-name))
              (if (= ?nth-field-width 38) then
                  (format t "%38s" ?nth-field-name))
              (if (= ?nth-field-width 39) then
                  (format t "%39s" ?nth-field-name))
              (if (= ?nth-field-width 40) then
                  (format t "%40s" ?nth-field-name))
            (bind ?field-index (+ ?field-index 1)))
     (printout t crlf)
     (retract ?f1))

(defrule display-records
     (declare (salience 5050))
     (display-records ?table-name)
     (?table-name field-width $?field-widths)
     (?table-name record ?index $?items)
=>
     (bind ?number-fields (length $?items))
     (bind ?field-index 1)
     (while (<= ?field-index ?number-fields)
              (bind ?nth-field-name (str-cat (nth ?field-index $?items)))
              (bind ?nth-field-width (nth ?field-index $?field-widths))
              (if (= ?nth-field-width 1) then
                  (format t "%1s" ?nth-field-name))
              (if (= ?nth-field-width 2) then
                  (format t "%2s" ?nth-field-name))
              (if (= ?nth-field-width 3) then
                  (format t "%3s" ?nth-field-name))
              (if (= ?nth-field-width 4) then
                  (format t "%4s" ?nth-field-name))
              (if (= ?nth-field-width 5) then
                  (format t "%5s" ?nth-field-name))
              (if (= ?nth-field-width 6) then
                  (format t "%6s" ?nth-field-name))
              (if (= ?nth-field-width 7) then
                  (format t "%7s" ?nth-field-name))
              (if (= ?nth-field-width 8) then
                  (format t "%8s" ?nth-field-name))
              (if (= ?nth-field-width 9) then
                  (format t "%9s" ?nth-field-name))
              (if (= ?nth-field-width 10) then
                  (format t "%10s" ?nth-field-name))
              (if (= ?nth-field-width 11) then
          (format t "%11s" ?nth-field-name))
     (if (= ?nth-field-width 12) then
          (format t "%12s" ?nth-field-name))
     (if (= ?nth-field-width 13) then
          (format t "%13s" ?nth-field-name))
     (if (= ?nth-field-width 14) then
          (format t "%14s" ?nth-field-name))
     (if (= ?nth-field-width 15) then
          (format t "%15s" ?nth-field-name))
     (if (= ?nth-field-width 16) then
          (format t "%16s" ?nth-field-name))
     (if (= ?nth-field-width 17) then
          (format t "%17s" ?nth-field-name))
     (if (= ?nth-field-width 18) then
          (format t "%18s" ?nth-field-name))
     (if (= ?nth-field-width 19) then
          (format t "%19s" ?nth-field-name))
     (if (= ?nth-field-width 20) then
          (format t "%20s" ?nth-field-name))
     (if (= ?nth-field-width 21) then
          (format t "%21s" ?nth-field-name))
     (if (= ?nth-field-width 22) then
          (format t "%22s" ?nth-field-name))
     (if (= ?nth-field-width 23) then
          (format t "%23s" ?nth-field-name))
     (if (= ?nth-field-width 24) then
          (format t "%24s" ?nth-field-name))
     (if (= ?nth-field-width 25) then
          (format t "%25s" ?nth-field-name))
     (if (= ?nth-field-width 26) then
          (format t "%26s" ?nth-field-name))
     (if (= ?nth-field-width 27) then
          (format t "%27s" ?nth-field-name))
     (if (= ?nth-field-width 28) then
          (format t "%28s" ?nth-field-name))
     (if (= ?nth-field-width 29) then
          (format t "%29s" ?nth-field-name))
     (if (= ?nth-field-width 30) then
          (format t "%30s" ?nth-field-name))
     (if (= ?nth-field-width 31) then
          (format t "%31s" ?nth-field-name))
     (if (= ?nth-field-width 32) then
          (format t "%32s" ?nth-field-name))
     (if (= ?nth-field-width 33) then
          (format t "%33s" ?nth-field-name))
     (if (= ?nth-field-width 34) then
          (format t "%34s" ?nth-field-name))
     (if (= ?nth-field-width 35) then
          (format t "%35s" ?nth-field-name))
     (if (= ?nth-field-width 36) then
          (format t "%36s" ?nth-field-name))
     (if (= ?nth-field-width 37) then
          (format t "%37s" ?nth-field-name))
     (if (= ?nth-field-width 38) then
          (format t "%38s" ?nth-field-name))
     (if (= ?nth-field-width 39) then
          (format t "%39s" ?nth-field-name))
     (if (= ?nth-field-width 40) then
               (format t "%40s" ?nth-field-name))
            (bind ?field-index (+ ?field-index 1)))
     (printout t crlf))

(defrule retract-display-records
     (declare (salience 5000))
     ?f1<-(retract-display-records)
     ?f2<-(display-records ?table-name)
=>
     (retract ?f1 ?f2))

;************************************************************************
;INSERT file:  Inserts records contained in the file specified by
;              <file-name> into the table specified by <table-name>.
;The file is of the format:
;              <field-list> <return>
;              <record-items-list> <return>
;              <other-records>
;<field-list> is a list of field names separated by spaces and terminated
;by <return>.  <record-items-list> is a list of record items separated
;by spaces and terminated by <return>.  Any number of records using the
;same format may be included.  Multi-word field names and record items must
;use underscores (_)'s or dashes (-)'s to separate words.
;Usage:  insert into <table-name> file <file-name>
;************************************************************************
(defrule insert-file-record
     (declare (salience 5000))
     ?f1<-(insert into ?table-name file ?file-name)
=>
     (open (str-cat ?table-name ".tbl") file-tag0 "r")
     (bind ?table-field-names (readline file-tag0))
     (close file-tag0)
     (bind ?number-fields (length (str-explode ?table-field-names)))
     (open (str-cat ?file-name) file-tag1 "r")
     (open (str-cat ?table-name ".tbl") file-tag2 "a")
     (bind ?file-field-names (readline file-tag1))
     (if (neq ?file-field-names EOF) then
         (bind ?file-input 0)
         (while (neq ?file-input EOF)
                (bind ?file-input (readline file-tag1))
                (if (neq ?file-input EOF) then
                    (bind ?field-index 1)
                    (bind ?record-string "record")
                    (while (<= ?field-index ?number-fields)
                           (bind ?next-field-name (nth ?field-index
                            (str-explode ?table-field-names)))
                           (bind ?position-in-file (member
                            ?next-field-name (str-explode
                                                 ?file-field-names)))
                           (if (> ?position-in-file 0) then
                               (bind ?record-string (str-cat
                                ?record-string " " (nth
                                ?position-in-file (str-explode
                                ?file-input))))
                              else
                               (bind ?record-string (str-cat
                                ?record-string " *")))
                           (bind ?field-index (+ ?field-index 1)))
```

```
                   (assert (do-load-table ?table-name))
                   (printout file-tag2 (mv-delete 1
                    (str-explode ?record-string)) crlf))))
      (close file-tag2)
      (close file-tag1)
      (retract ?f1))

;*********************************************************************
;WHERE:  Parses the conditional list and returns the records that satisfy
;        the conditional list.  The records are asserted in the format
;        ($?junk record-seq <record-numbers>) into the fact-list.
;        <record-numbers> correspond to record indexes.  $?junk is to be
;        ignored.
;Usage:  in <table-name> where <conditional-list>
;<table-name> refers to a table currently in the fact-list.
;<conditional-list> follows the format:  <field-name> <comparator> <value>
;A rule extracting values from ($?junk record-seq <record-numbers>) must
;have a salience value of at least 6000, or the rule will not extract
;the values before the fact is retracted from the fact-list.
;*********************************************************************
(defrule where-or
      (declare (salience 5300))
      ?f1<-($?or in ?table-name where $?conditional1 or $?conditional2)
=>
      (assert (or in ?table-name where $?conditional1))
      (assert (or in ?table-name where $?conditional2))
      (retract ?f1))

(defrule union-record-seq
      (declare (salience 5200))
      ?f1<-(or $?and1 record-seq $?seq1)
      ?f2<-(or $?and2 record-seq $?seq2)
=>
      (if (neq $?seq1 $?seq2) then
          (str-assert (str-cat "record-seq " (str-implode $?seq1) " "
                         (str-implode $?seq2)))
          (retract ?f1 ?f2))
      (if (and (eq $?seq1 $?seq2) (neq $?and1 $?and2)) then
          (retract ?f2)))


(defrule where-and
      (declare (salience 5250))
      ?f1<-($?and/or in ?table-name where $?conditional1 and
             $?conditional2)
=>
      (if (eq $?conditional1 $?conditional2) then
          (assert ($?and/or in ?table-name where $?conditional1)) else
          (assert ($?and/or and in ?table-name where $?conditional1))
          (assert ($?and/or and in ?table-name where $?conditional2)))
      (retract ?f1))

(defrule intersection-record-seq
      (declare (salience 5400))
      ?f1<-($?or1 and record-seq $?seq1)
      ?f2<-($?or2 and record-seq $?seq2)
=>
      (if (neq $?seq1 $?seq2) then
        (bind ?seq1-length (length $?seq1))
        (bind ?seq-index 1)
        (bind ?string "record-seq")
        (while (<= ?seq-index ?seq1-length)
               (bind ?next-item (nth ?seq-index $?seq1))
               (if (> (member ?next-item $?seq2) 0) then
                   (bind ?string (str-cat ?string " " ?next-item)))
               (bind ?seq-index (+ ?seq-index 1)))
        (str-assert (str-cat (str-implode $?or1) " " ?string))
        (retract ?f1 ?f2))
    (if (and (eq $?seq1 $?seq2) (neq $?or1 $?or2)) then
        (retract ?f2)))

(defrule seed-logical-sequence
    (declare (salience 5300))
    ($?operators in ?table-name converted-where ?field-name ?logical
     ?value)
=>
    (assert (logical-seq))
    (assert (cycle-index 1)))

(defrule convert-where
    (declare (salience 5300))
    ?f1<-($?operators in ?table-name where ?field-name ?logical
          ?value)
=>
    (assert ($?operators in ?table-name converted-where ?field-name
            ?logical ?value))
    (retract ?f1))

(defrule where-logical
    (declare (salience 5350))
    ($?operators in ?table-name converted-where ?field-name ?logical
     ?value)
    ?f2<-(cycle-index ?record-index)
    (?table-name record ?record-index $?items)
    (?table-name field-name $?field-line)
    ?f1<-(logical-seq $?sequence)
=>
    (retract ?f2)
    (bind ?new-cycle-index (+ ?record-index 1))
    (assert (cycle-index ?new-cycle-index))
    (bind ?field-index (member ?field-name $?field-line))
    (bind ?file-value (nth ?field-index $?items))
    (if (eq ?logical eq) then
              (if (eq ?file-value ?value) then
                  (retract ?f1)
                  (assert (logical-seq $?sequence ?record-index))))
    (if (eq ?logical neq) then
              (if (neq ?file-value ?value) then
                  (retract ?f1)
                  (assert (logical-seq $?sequence ?record-index))))
    (if (eq ?logical <) then
              (if (< ?file-value ?value) then
                  (retract ?f1)
                  (assert (logical-seq $?sequence ?record-index))))
    (if (eq ?logical <=) then
              (if (<= ?file-value ?value) then
                    (retract ?f1)
                    (assert (logical-seq $?sequence ?record-index))))
     (if (eq ?logical >) then
                (if (> ?file-value ?value) then
                    (retract ?f1)
                    (assert (logical-seq $?sequence ?record-index))))
     (if (eq ?logical >=) then
                (if (>= ?file-value ?value) then
                    (retract ?f1)
                    (assert (logical-seq $?sequence ?record-index)))))

(defrule clean-up-where-logical
     (declare (salience 5325))
     ?f1<-($?operators in ?table-name converted-where ?field-name
           ?logical ?value)
     ?f2<-(logical-seq $?sequence)
     ?f3<-(cycle-index ?record-index)
=>
     (retract ?f1 ?f2 ?f3)
     (assert ($?operators record-seq $?sequence)))
```